Broadview®
www.broadview.com.cn

LBS核心技术揭秘

LBS是移动应用的本质，如果你对此感兴趣，
那么本书将是你深入探索LBS世界的不二之选。

贾双成 编著

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

贾双成，江南大学硕士，现为阿里巴巴高级工程师，擅长于数据编译、数据挖掘的系统分析和架构设计，研究方向包括几何算法、数据编译、数据挖掘算法及应用。

曾发表专利、论文三十余篇。在研究算法之余，也喜欢涉猎管理学、哲学、心理学、历史等领域的知识。

LBS核心技术揭秘

贾双成 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是作者根据实际的软件研发经验编写而成的，它弥补了目前 LBS 核心技术领域的市场空白，内容均是作者在 LBS 领域中实际工程经验的总结。全书内容包括三部分：LBS 基础知识、技术架构和核心技术，内容涵盖 LBS 研发的所有关键技术，包括 GIS 知识、编程知识、技术架构、数据处理、数据挖掘、导航、显示、搜索、网络传输和后台服务。每章内容相对独立。

本书内容没有华而不实的泛泛之谈，每一部分内容对实际的代码开发都有很大的帮助，希望本书能成为 LBS 开发人员必备的一本案头书。

本书不是一本读完一遍就可以束之高阁的快餐读物，而是一本能解决 LBS 开发人员疑难问题的参考手册。希望本书能助你成为一名 LBS 开发的行家或快乐的程序员。

图书在版编目（CIP）数据

LBS 核心技术揭秘 / 贾双成编著. —北京：电子工业出版社，2015.7
ISBN 978-7-121-26214-2

Ⅰ. ①L… Ⅱ. ①贾… Ⅲ. ①地理信息系统—研究 Ⅳ. ①P208

中国版本图书馆 CIP 数据核字（2015）第 116148 号

策划编辑：孙学瑛
责任编辑：李利健
印　　刷：北京京师印务有限公司
装　　订：北京京师印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×980　1/16　印张：15.5　字数：297 千字
版　　次：2015 年 7 月第 1 版
印　　次：2015 年 7 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

献给我的儿子——阳明

小家伙，世界因你而不同！

前　言

只要打开智能手机，你就会发现 LBS 应用已成为主流，比如：美团、糯米、高德地图、陌陌等。但目前在市场上却没有一本书来揭示 LBS 的核心技术。在今天的图书市场上，虽然我们能看到很多关于网页开发、手机开发、语言编程或者地理信息系统（GIS）的书，但这些书对我们开发 LBS 的应用来说，都只是在描述一些“外壳”，并没有揭示出 LBS 本身内在的核心技术。

LBS 技术覆盖范围广，包含了多个领域的知识，这些领域不仅包括一些已经形成专业的领域，比如：GIS、计算几何、数据挖掘、图像处理、网络通信、图论、三维渲染等，还包括一些新兴的工程领域，比如：TMC、Web Service 等。

由于 LBS 技术覆盖的范围如此广泛，在工作中，我常常看到的一种情形是：很多同事是 GIS 技术出身，对 GIS 领域的知识很精通，但对计算机领域的知识却并不明白；而另一些同事是学计算机知识出身，对计算机很精通，但对 GIS 并不了解。由于大家的知识结构是如此不完整，所以在实际工作中，普遍存在学习成本高、进步慢的状况。我经常看到的一种情况是：不管是什么专业的人，只要新进入 LBS 行业，往往不能立刻工作，要经历极长的学习周期。即使是 LBS 行业的老兵，也往往跟不上 LBS 已经或者正在发生的翻天覆地的变化，很多人的知识仍然停留在 GIS 时代，知识面严重受限。

看起来，LBS 已经成为一个包罗万象，谁都说不清楚的领域。但是作者在实际工作中发现，其实真正对 LBS 工作有用的知识并不是很多，因为许多同事可能就是在靠一两个算法过日子，只不过这些并不神秘的算法却被“封印”了而已。

我进入 IT 领域已经十年了，进入 LBS 领域也有五年。在 LBS 领域，我也经历了一个漫长的黑暗的摸索过程，所以，我希望把我在工作中常用到的技术系统地写出来，以供 LBS 领域的朋友参考。希望新人能够快速成长，少走弯路；也希望 LBS 领域的老人能借此书总结或归纳自己的知识。

现在的 LBS 浪潮只是一个开始，仅是揭开了一个序幕而已。我相信，在 LBS 领

域，肯花时间来系统地学习一点知识是值得的。因为不管我们是得过且过，还是选择做点什么，时间总是要流走的。将时间投资在一个能长期蓬勃向上发展的领域是值得的。

一本参考书需要经过许多人的使用和反馈之后才能变得更加完善。由于本书作者的经验和时间有限，书中的错误和纰漏难免，敬请读者不吝指正。

贾双成

致　谢

感谢在以往十年的工作中，所有曾与我在一起工作的同事，与你们在一起工作是本书的灵感来源。

感谢高德所提供的良好开发环境，以及俞永福总裁对本书的支持。

感谢叶旻提供了一个对开发人员来说比较简单的环境，让我能够静下心来总结以往的知识，并能一直保持对新知识的汲取和学习。

在本书的内容审核上，许多同事都提出了一些宝贵的建议，这些同事包括陈岳、王奇、逯志欣、刘义洲、胡守兴、朱玟征、张志斌、邹剑章。

感谢正在阅读本书的读者。我知道，你正走在成为未来 LBS 中坚人物的路上。这一路上，你要知道，你并不孤独，阅读本书，我即与你同在。

贾双成

目　录

第 1 部分　LBS 基础知识

第 2 部分　LBS 常用技术架构

第 3 部分 LBS 各模块的核心技术

附 录

第 1 部分　LBS 基础知识

基础是经典的代名词

第 1 章

基于位置的服务

智能手机及移动应用深刻地改变了人们的生活，而 LBS（Location-Based Services，基于位置的服务，简称 LBS）正是移动应用区别于 PC 应用的主要特征。

1.1 背景

每一项新技术的出现都会产生新的商机，重大的科技创新尤其如此。智能手机作为这种革命性技术的代表，已经对人们的日常生活产生了巨大的影响。如今，不管是在地铁上，还是在餐馆里吃饭，低头看手机的人群已经占了大多数。

智能手机的出现不仅改变了人与人的交流方式（比如：微信、陌陌），也改变了人的购物方式（比如，O2O 的团购 App：美团、糯米等），还改变了人的工作方式（比如：GPS 定位下的物流或销售人员的管理）、娱乐和休闲方式（比如：各种智能手机的游戏应用）。在这些智能手机所带来的改革浪潮中，基于位置的服务已经拔得头筹。

1.2 含义

LBS（基于位置的服务）通过无线电通信网络（如通信运营商的 GSM 网、CDMA 网或网络 Wi-Fi）或外部定位方式（其中，用 GPS 定位来获得用户位置的方法是目前的主流方法）获取移动终端用户的位置信息（某种地理坐标），在 GIS（Geographic Information System，地理信息系统）平台的支持下，为用户提供的某种服务（比如：O2O、社交、游戏等）。

LBS 通过一组定位技术获得移动终端的位置信息（如经纬度坐标数据），从而实现各种与位置相关的服务。LBS 实质上是一种与空间位置有关的服务的统称。

一般情况下，LBS 系统由以下模块组成。

- 空间位置获取（定位平台）；
- 地理信息系统（GIS）；

- 业务服务；
- 信息传送；
- 移动智能终端。

其中，各模块的含义如下。

（1）空间位置获取系统

该模块主要通过定位技术获取移动客户准确的地理位置，这里的地理位置数据是 LBS 系统的基础。该模块通常由一些定位模块（采用 GPS 或者 Wi-Fi，或者移动网络基站定位）构成。

（2）地理信息系统（GIS）

该模块一般体现为地图，可以自建，也可以使用第三方的大型地图服务商（如高德地图、百度地图等）所提供的 GIS 服务。GIS 是整个位置服务系统的基础，负责将移动终端的地理数据信息转换成地图中可视化位置的功能。一般情况下，我们从定位系统中只能获取到终端的三维地理空间坐标，这种数据只有通过 GIS 的处理，才能为业务服务系统所用。得到客户的地理位置信息也就相当于得到了客户的位置，只有得到了客户的位置，才能向客户提供相应的 LBS 服务。

（3）业务服务系统

该模块为客户提供具体的业务服务。根据不同的市场细分，业务服务系统可以为不同类型的客户提供不同的服务，如为时尚青年提供基于位置的游戏、聊天、交友服务，为家庭客户、商务人士提供移动保姆、交通导航、商业广告服务，为行业用户提供车辆调度、紧急救援、物流配送服务等。

另外，业务服务系统还负责隐私管理、用户认证管理、业务管理和计费管理等功能。

业务服务系统使移动客户可以获取他所需要的服务，如客户需要通过定位服务查询附近有哪些著名的花店、酒家信息。周围的这些花店或者酒家信息往往是业务服务提供系统的合作商家，其具体信息已事先录入业务系统，并与自建或第三方的地理信息系统相关联。以用户在团购网站搜索酒家为例，当用户在搜索某个酒家时，业务提供方可以用地理信息系统来展示酒家的位置，也可以借助地理信息系统来展示用户距离某酒家的距离。

（4）信息传送系统

该模块是指客户和内容提供商之间的网络传送平台。目前比较成熟的传送平台是无线网络或移动运营商的 2G/3G/4G 网络。

（5）移动智能终端

该模块是用户唯一接触的部分，手机、Pad 均可为 LBS 的用户终端。在信息化的现代社会，由于智能手机有完善的图形显示能力、良好的通信端口、友好的用户界面和完善的输入方式（键盘控制输入、手写板输入、语音控制输入等），且因为便携性出众。因此，智能手机已成为个人 LBS 终端的首选。

1.3 包含的领域

LBS 服务包含的领域如下。

- 物流（榜样企业：顺丰、沃尔玛）；
- O2O（榜样企业：美团、淘点点）；
- 拼车（榜样企业：快的、嘀嘀、神州租车）；
- 旅游；
- 导航（榜样企业：高德地图）；
- 社交（榜样企业：陌陌）；
- 游戏等。

具体地说，目前国内流行的 LBS 服务已数不胜数，包括：高德地图、百度地图、微信、美团、街旁等。打开如今的智能手机，我们可以发现，包含 LBS 功能的应用已经成为主流，很少有手机应用中没有包含 LBS 功能。例如，一款典型的手机中包含的 LBS 应用可能如下。

- 高德地图（主流导航应用）；
- 微信（内含 LBS 应用：摇一摇、附近的人等）；
- 58 同城；
- 陌陌；
- 美团、糯米等。

1.4 展望

在最近两三年，已经出现明显的趋势，即 LBS 正在重塑所有的应用。可以想象，在不远的未来会有以下变化。

- 未来的淘宝或者天猫的 O2O 模块中的商家一定会按照地理位置来重塑应用；
- 未来的广告系统也会按照地理系统来重塑应用，从而使用户走到某个商家周围时能看见周围的优惠信息；
- 未来的社交系统，特别是陌生人社交，或者熟人社区社交，一定会按照区域来组织，如同 Yik Yak（美国一款具有定位功能的匿名留言板产品）等正在探索的。

……

第 2 章

基础知识

测绘行业作为一个古老的行业，在电子化的网络时代，焕发了强大的生机。特别是在智能手机日益普及的今天，测绘行业正在计算机的帮助下重新变革自己，融入这个移动时代。

2.1 地图、测绘及坐标系

地球就在我们脚下。一直以来，我们的各种活动构成了我们对世界的认识，即形成我们的世界观。每个人对世界的认识并不相同，如果把人们认识世界的相同部分抽象出来，就形成了“地图”。

地图关于地球的学问，是 LBS 的基础。人们利用坐标系来建立地图结构，通过采集来进行地图数据制作。

2.1.1 地图和测绘的演变

1. 地图的演变

早在甲骨文时期，人们就利用文字留下了最早的关于地理的资料。现已发现的十几万块甲骨文卜辞材料所记录的关于地理内容十分可观，可以说是中国地理知识记录史的第一篇。

甲骨文中的地名有自然的山河名称，也有多样的风向说明，而更值得注意的是人文地理的东西。史学家把早期模糊的记载称为“史影”，那么，在支离残缺的甲骨卜辞中，不但有人文的“史影”，也有人文的“地影”。对人文的“地影”而言，卜辞专家如王国维、郭沫若、陈梦家、李学勤等都进行过研究和推断，使我们对商代的人文地理态势有了稍微具体的认识。卜辞中最常见的人文地理内容有城、邑、边鄙（郊区）、商王的田猎区、四土、邦方（方国部族）等，这几样东西构成了商代人文地理的主要框架。

甲骨文材料证明了商代已经出现大地域国家的早期特征，而国家领土只要大到

一定程度，就会出现所谓中央与地方的关系之类的问题。中国古代常说“王畿千里”，其中，“王畿”可以理解为“中央”，国土若超过了 500km，就有了“地方”。随着领土的扩大，国家机器要建立一套管理控制大地域的办法，具体地说，就是“中央”管控大量“地方”的办法，地理的政治内容因此出现。

世界上最早的保存完整的历史图集是在河南洛阳编绘成的。据历史记载，春秋战国时期，我国出现了地理名著《禹贡》，它把全国分为九州，然后分州叙述各地的山川、湖泊、土壤、物产、田赋、贡品和交通等情况。

到了魏晋时期，因为年代久远，《禹贡》中所记载的内容已发生变化，于是，裴秀便参考收集到的历史地图，结合晋朝的行政区划，在详细考证古今地名、山川和疆域的基础上，绘制了《禹贡地域图十八篇》。《禹贡地域图十八篇》不仅绘有新的郡国、府县政区划分和居民地的位置，而且把古代的九州、历史上各王朝曾经举行会议的会址、签订条约的地点和古地名等，皆一一标示出来。《禹贡地域十八篇》是当时中国最精详的历史地图集。在他之前不久，有人绘制了一幅《天下大图》，规模宏大，据说“用缣八十匹”，这在当时世界上是绝无仅有的，但是不便携带、阅览和保存。于是裴秀运用制图六体的方法“以一分为十里，一寸为百里”的比例尺，缩绘成为约一百八十万分之一的《地形方丈图》，这样就简便了。他所缩绘的《地形方丈图》一直流传应用了好几百年，不仅在我国地图史上具有划时代的意义，而且在世界地图史上也占有突出的位置。

而在印度，由于印度人对世界的想象充满了宗教和唯美色彩，所以在印度的宗教文化中，整个世界被描绘成莲花的形状，一片花瓣就代表一块陆地。

在北美，目前发现最早的一幅地图被画在一张皮革上，在所有的北美原住居民的地图中，河流都扮演着非常重要的角色。古印加人留下的地图充满了不解之谜，他们的道路系统从何而来？又被什么人所继承？我们仅能知道的是，在印加文明之前，安第斯山脉的居民已经有了自己的一套理念去更好地表达空间知识。

最有趣的是波利尼西亚人的地图，这些天生的航海家仅凭几根小木棍就能表达出他们获取的新的航海信息——根据距离不同而导致的岛屿间的形状变化来调整他们的位置，这些木棍被扎成只有他们自己能看懂的“航海图”，被记录下来。更神奇的是，他们出海远行时并不需要随身携带这些“航海图”，而是将它们熟记于心。

在欧洲，希腊人在大约公元前 6 世纪就提出地球应当是一个球体。伟大的托勒密是“西方地图学奠基者”，也是现代地图学的鼻祖。他发明了以新投影方法来绘

制世界地图。

中世纪的地图学几乎完全被宗教寰宇观所取代。这些图把世界画成一个大圆盘，耶路撒冷位于圆盘的中心，圆的南部一横分别是尼罗河与顿河，中间一竖为地中海，构成丁字形水体，并分隔为亚、欧、非三个大陆。这类地图数量不少，但几乎千篇一律。这种状况一直延续到15世纪。

到了16世纪，由于航海技术的发展，人类对整个世界的认识都进入了全新的探索阶段。欧洲人的远航拯救了地图文化。哥伦布发现美洲，达·加马开辟印度新航路，麦哲伦完成环球航行。随着欧洲航海事业日益发达，探险家的足迹遍布非洲、美洲和亚洲。

欧洲人为新大陆绘制地图的过程也使得本土地图绘制的技术得到不断提高。对地图的需求随着印刷术的出现而得到了扩展；科技的发展也为更加精确的绘图测量开辟了道路。17世纪，在全球贸易中领先的荷兰人同时成为全世界领先的地图绘制者。

2. 测绘手段的演变

在古代，测绘手段很有限，往往利用人本身的感觉来作为世界的感知与测量，由于没有坐标系和参照物的概念，所以很多记载都陷于虚幻，例如，《山海经》。

魏晋时期的裴秀在《禹贡地域十八篇》序言中提出了《制图六体》："制图之体有六焉，一曰分率，所以辩广轮之度也；二曰准望，所以正彼此之体也；三曰道里，所以定所由之数也；四曰高下，五曰方邪，六曰迂直，此三者，各因地而制宜，所以校夷险之异也。"这里，裴秀提出了绘制地图应该遵循六条基本原则。其中：分率就是比例尺，准望就是方位，道里即是距离，高下就是相对高度，方邪就是地面坡度起伏，迂直就是实地的高低起伏距离与平面图上距离的换算。今天地图学上所应考虑的主要因素，裴秀几乎都考虑到了。但是限于当时的测绘技术条件，所以当时的地图仍是粗糙的。由于地图的粗糙，所以关于地理的神话传说在人们的脑海中仍有其市场。

不同文化的交融是世界文明发展的推动力量。中国的传统测绘在融合了西方测绘术后，也跃上了一个新台阶。在传播西方测绘术的先驱者中，徐光启是功绩最为卓著的。

徐光启是明代著名科学家，他师从来到中国的意大利传教士利马窦，学习天文、

历算、测绘等。资质聪慧的徐光启很快得其要旨，并有所创造。在徐光启等中国学者的一再要求和推动下，外国传教士才同意翻译外国科技著作，向中国人介绍西方的测绘技术。明朝后期问世的测绘专著和译著大多与徐光启有关。徐光启和利马窦合译了《几何原本》和《测量法义》，也与熊三拔合译了《简平仪说》。徐光启认为，《几何原本》是测算和绘图的数学基础，力主翻译。为了融通东西，他撰写了《测量异同》，考证中国测量术与西方测量术的相同点和不同点。他主持编写了《测量全义》，这是集当时测绘学术之大成的力作，内容丰富，涉及面积、体积测量和有关平面三角、球面三角的基本知识以及测绘仪器的制造等。

徐光启还身体力行，积极推进西方测绘术在实践中的应用，1610 年，他受命修订历法。他认为，修历法必须测时刻、定方位、测子午、测北极高度等，于是要求成立采用西方测量术的西局和制造测量仪器。此次仪器制造的规模在我国测绘史上是少见的，共制造象限大仪、纪限大仪、平悬浑仪、转盘星晷、候时钟、望远镜等 27 件。利用新制仪器进行了大范围的天象观测，取得了一批实测数据，其中载入恒星表的有 1347 颗星，这些星都标有黄道、赤道经纬度。总之，无论在理论上还是在实践上，徐光启都算得上传播西方测绘术最卓越的先驱者。

时光的车轮演进到今天，测绘技术达到了一个前所未有的新高峰。比如，目前的测绘仪器有：三维激光扫描仪、水准仪、经纬仪、全站仪、GPS 接收机、GPS 手持机、超站仪、陀螺仪、求积仪、钢尺、秒表等，甚至，相机也成了测绘中使用的仪器。

2.1.2 地图采集、制作

1. 目前的采集和制作技术

在当前的地图采集中，常见的情况如下。

- 得到新的道路形状的方法是：利用道路采集车或出租车众包数据进行道路的形状点的采集，之后，利用人工来对形状点进行化简和修正，从而对道路数据进行更新；
- 得到新的 POI（兴趣点）的方法是：外业人员到商铺门口去采集，并利用 GPS 确定位置，内业人员将外业采集得到的数据更新到地理数据库；
- 得到各种限速或警告、道路指示牌的方法是：内业人员利用外业人员的道路采集车得到的录像回放来确定道路标志的位置；

- 得到 3D 建筑物的方法是：利用人工对拍摄到的建筑物数据进行建模；
- 得到卫星图的方法是通过卫星；
- 得到高精度 DTM 的方法一般是通过测绘飞机。

可以看出，目前的测绘手段仍然较多地依赖于人工，其精度不够高，成本也较大。

2. 采集、制作技术的未来发展

可以想象，未来的地图数据采集会将人工的部分替代成计算机程序实现。比如：

- 利用飞机航拍得到 3D 的建筑物模型；
- 利用航拍飞机、高精度的 DTM、众包的出租车回传数据得到道路的高精度的曲率和坡度；
- 对高精度采集车得到的数据利用程序进行处理，经过去噪、插值拟合、化简等步骤，最终得到所需道路的形状点、曲率或坡度信息；
- 利用对图像进行识别来得到各种警告、速度限制等文字信息。

……

2.1.3 地理坐标系

众所周知，地球是一个球体，我们每一个人所在的位置都在地球上占据了一个特定的空间。

如果要精确地表达我们的位置，就需要建立坐标系。常用的地图坐标系有两种，即地理坐标系（可认为是球体坐标）和投影坐标系（可认为是平面坐标）。

地理坐标系（或称为大地坐标）是以经纬度为单位的地球坐标系统，地理坐标系中有两个重要的部分，即地球椭球体（ellipsoid）和大地基准面（datum）。由于地球表面的不规则性，它不能用数学公式来表达，也就无法实施运算，所以必须找一个形状和大小都很接近地球的椭球体来代替地球，这个椭球体被称为地球椭球体，在 LBS 领域常用的椭球体是 WGS84 椭球体。大地基准面指当前参考椭球与 WGS84 参考椭球间的相对位置关系（共有 3 个平移参数、3 个旋转参数和 1 个缩放参数），可以用其中 3 个、4 个或者 7 个参数来描述它们之间的关系。在具体实现上，地面点 P 的位置用大地经度 L、大地纬度 B 和大地高 H 表示。当点在参考椭球面上时，仅用大地经度和大地纬度表示。大地经度是通过该点的大地子午面与起始大地子午面

之间的夹角，大地纬度是通过该点的法线与赤道面的夹角，大地高是地面点沿法线到参考椭球面的距离。

投影坐标系是利用一定的数学法则把地球表面上的经纬线网表示到平面上，属于平面坐标系。数学法则指的是投影类型。目前我国普遍采用的是高斯–克吕格投影，在英美国家称为横轴墨卡托投影（Transverse Mercator）。高斯–克吕格投影的中央经线和赤道为互相垂直，分带标准分为 3 度带和 6 度带。美国编制世界各地军用地图和地球资源卫星相片所采用的全球横轴墨卡托投影（UTM）是高斯-克吕格投影的一种变型。高斯–克吕格投影的中央经线长度比等于 1，UTM 投影规定中央经线长度比为 0.9996。

如上所述，LBS 中常用的几种坐标系有：地理坐标系（经纬度坐标系或 WGS84 经纬度坐标）和投影坐标系（高斯坐标系、UTM 坐标系）。下面将详细描述各种坐标系的特点。

1. 经纬度坐标系

这是在中国最常用的坐标系。首先将地球抽象成一个规则的逼近原始自然地球表面的椭球体，称为参考椭球体，然后在参考椭球体上定义一系列的经线和纬线构成经纬网，从而达到通过经纬度来描述地表点位的目的。需要说明的是，经纬地理坐标系不是平面坐标系，因为度不是标准的长度单位，不可用其直接量测面积长度。

经纬度通常分为天文经纬度、大地经纬度和地心经纬度。常用的经度和纬度是地心经纬度，即从地心到地球表面上某点的测量角，通常以度或百分度为单位来测量该角度。地心经纬度示意图如图 2-1 所示。

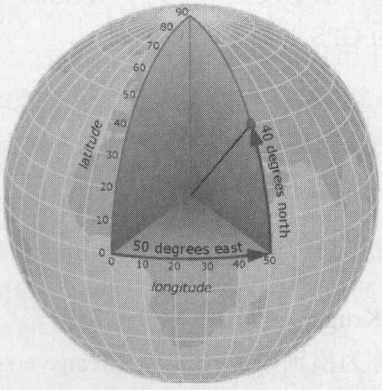


图 2-1　经纬度示意图

在球面系统中，水平线（或东西线）是等纬度线或纬线。垂直线（或南北线）是等经度线或经线。这些线包络着地球，构成了一个称为经纬网的格网化网络。

位于两极点中间的纬线称为赤道，它定义的是零纬度线。零经度线称为本初子午线。对绝大多数地理坐标系来说，本初子午线是指通过英国格林尼治的经线。其他国家/地区使用通过伯尔尼、圣菲波哥大和巴黎的经线作为本初子午线。经纬网的原点（0，0）定义在赤道和本初子午线的交点处。这样，地球就被分为四个地理象限，它们均基于与原点所构成的罗盘方位角。南和北分别位于赤道的下方和上方，而西和东分别位于本初子午线的左侧和右侧。

通常，经度和纬度值以十进制度为单位或以度、分和秒（DMS）为单位进行测量。纬度值相对于赤道进行测量，其范围是 –90°（南极点）到+90°（北极点）。经度值相对于本初子午线进行测量，其范围是–180°（向西行进时）到 180°（向东行进时）。如果本初子午线是格林尼治子午线，则对位于赤道南部和格林尼治东部的澳大利亚来说，其经度为正值，纬度为负值。

由于经纬度可以是采样当地坐标系计算出来的，而这种地方经纬度可能并不适用于其他地方。所以，地方性的经纬度坐标系在使用中急需升级，这就形成了目前全球通用的 WGS84 经纬度坐标系。

WGS84（World Geodetic System 1984）经纬度是采用 WGS84 坐标系所换算出来的经纬度，该坐标系是目前 GPS 所采用的坐标系统，是一个地心地固坐标系统。

WGS84 是为 GPS 全球定位系统使用而建立的坐标系统。通过遍布世界的卫星观测站观测到的坐标建立，其初次 WGS84 的精度为 1～2m。在 1994 年 1 月 2 日，通过 10 个观测站在 GPS 测量方法上改正，得到了 WGS84（G730），其中，G 表示由 GPS 测量得到，730 表示为 GPS 时间第 730 个周。

WGS84 坐标系的几何意义是：坐标系的原点位于地球质心，*Z* 轴指向（国际时间局）BIH 1984.0 定义的协议地球极（CTP）方向，*X* 轴指向 BIH 1984.0 的零度子午面和 CTP 赤道的交点，*Y* 轴通过右手规则确定。

2. 高斯坐标系

高斯–克吕格（Gauss-Kruger）投影简称“高斯投影”，又名“等角横切椭圆柱投影”，在英美国家又被称为横轴墨卡托投影（Transverse Mercator），是地球椭球面和平面间正形投影的一种。德国数学家、物理学家、天文学家高斯（Carl Friedrich

Gauss，1777—1855）于 19 世纪 20 年代拟定，后经德国大地测量学家克吕格（Johannes Kruger，1857—1928）于 1912 年对投影公式加以补充，由此得名。该投影按照投影带中央子午线投影为直线且长度不变和赤道投影为直线的条件，确定函数的形式，从而得到高斯-克吕格投影公式。投影后，除中央子午线和赤道为直线外，其他子午线均为对称于中央子午线的曲线。设想用一个椭圆柱横切于椭球面上投影带的中央子午线，按上述投影条件，将中央子午线两侧一定经差范围内的椭球面正形投影于椭圆柱面。将椭圆柱面沿过南北极的母线剪开展平，即为高斯投影平面。取中央子午线与赤道交点的投影为原点，中央子午线的投影为纵坐标 X 轴，赤道的投影为横坐标 Y 轴，构成高斯–克吕格平面直角坐标系。

高斯–克吕格投影在长度和面积上变形很小，中央经线无变形，自中央经线向投影带边缘，变形逐渐增加，变形最大之处在投影带内赤道的两端。由于其投影精度高、变形小，而且计算简便（各投影带坐标一致，只要算出一个带的数据，其他各带都能应用），因此，在大比例尺地形图中应用可以满足军事上的各种需要，能在图上进行精确的测量计算。

高斯–克吕格投影按一定经差将地球椭球面划分成若干投影带，这是高斯投影中限制长度变形的最有效的方法。分带时既要控制长度变形使其不大于测图误差，又要使带数不致过多以减少换带计算工作，据此原则将地球椭球面沿子午线划分成经差相等的瓜瓣形地带，以便分带投影。通常按经差 6°或 3°分为六度带或三度带。六度带自 0°子午线起每隔经差 6°自西向东分带，带号依次编为第 1，2，…，60 带。三度带是在六度带的基础上分成的，它的中央子午线与六度带的中央子午线和分带子午线重合，即自 1.5°子午线起每隔经差 3°自西向东分带，带号依次编为三度带第 1，2，…，120 带。我国的经度范围西起 73°、东至 135°，可分成六度带 11 个，各带中央经线依次为 75°，81°，87°，…，117°、123°、129°、135°，或三度带 22 个。六度带可用于中小比例尺（如 1∶250000）测图，三度带可用于大比例尺（如 1∶10000）测图，城建坐标多采用三度带的高斯投影。

高斯–克吕格投影按分带方法各自进行投影，故各带坐标成独立系统。以中央经线投影为纵轴（X），赤道投影为横轴（Y），两轴交点即为各带的坐标原点。纵坐标以赤道为零起算，赤道以北为正，以南为负。我国位于北半球，纵坐标均为正值。横坐标如以中央经线为零起算，中央经线以东为正，以西为负，横坐标出现负值，使用不便，故规定将坐标纵轴西移 500km 当作起始轴，凡是带内的横坐标值均加 500km。由于高斯-克吕格投影每一个投影带的坐标都是对本带坐标原点的相对值，

所以各带的坐标完全相同，为了区别某一坐标系统属于哪一带，在横轴坐标前加上带号，如（4231898m，21655933m），其中 21 为带号。

3. UTM 坐标系

UTM 投影全称为“通用横轴墨卡托投影”，由于横轴墨卡托投影就是高斯-克吕格投影，所以 UTM 投影是高斯-克吕格投影的一种变型。UTM 投影是等角横轴割圆柱投影（高斯-克吕格为等角横轴切圆柱投影），圆柱割地球于南纬 80°、北纬 84°两条等高圈。该投影将地球划分为 60 个投影带，每带经差为 6°，已被许多国家作为地形图的数学基础，在欧洲地图中应用普遍。UTM 投影与高斯投影的主要区别在于南北格网线的比例系数上，高斯-克吕格投影的中央经线投影后保持长度不变，即比例系数为 1，而 UTM 投影的比例系数为 0.9996。UTM 投影沿每一条南北格网线比例系数为常数，在东西方向则为变数，中心格网线的比例系数为 0.9996，在南北纵行最宽部分的边缘上距离中心点大约 363km，比例系数为 1.00158。

国外某些地图编辑软件如 ARCGIS/MAPINFO，或国外仪器的配套软件如多波束的数据处理软件等，往往不支持高斯-克吕格投影，但支持 UTM 投影。因此，常有把 UTM 投影坐标当作高斯-克吕格投影坐标提交的现象。

高斯－克吕格投影与 UTM 投影可近似采用“Xutm=0.9996×X 高斯、Yutm=0.9996×Y 高斯”进行坐标转换。

2.2 编程基础知识

计算机的主要用途就是大量数据的存储和快速计算。大量数据的存储往往涉及大容量的硬盘和更好的压缩算法。为了实现快速地计算，本质上就是一个查找问题，而查找则依赖于排序和更好的数据结构。

2.2.1 排序方法

排序主要包含内部排序和外部排序。所谓内部排序（简称内排序），是指所有待排序内容都存储在内存的排序。所谓外部排序（简称外排序），是指有部分待排序的内容没有被写入内存，而是被存储在硬盘的排序。显然，内排序适用于规模在千万量级内的数据，外排序适用于大规模的数据。由于，外排序的操作需要和硬盘进行反复交互，所以效率要比内排序低。

内排序主要有三种方法：插入排序、选择排序和交换排序。

外排序主要是用归并排序。

除以上排序方法外，还有其他一些排序方法，比如：基数排序或者桶排序，其本质是将待排序的数据填入“键值已排序的 Hash 表”。所以，这种排序不是全新的有指导意义的方法，在 2.2.2 节（基础算法章节的数据结构部分）中也会有所涉及。

在待排序的文件中，若存在多个关键字相同的记录，经过排序后，这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是稳定的；若具有相同关键字的记录之间的相对次序发生改变，则称这种排序方法是不稳定的。

1. 插入排序

插入排序的基本思想是每步将一个待排序的记录按其排序码值的大小，插到前面已经排好的文件中的适当位置，直到全部插入完为止。

代码如下：

```
1. void Dir_Insert(int A[],int N) //直接插入排序
2. {
3. int j,t;
4. for(int i=1;i<N;i++)
5. {
6. t=A[i];
7. j=i-1;
8. while(A[j]>t)
9. {
10. A[j+1]=A[j];
11. j--;
12. }
13. A[j+1]=t;
14. }
15. }
```

插入排序的交换次数取决于逆序对的次数，所以插入排序对逆序对较少的情况特别有效。

```
void Dir_Insert(int A[],int N) //直接插入排序
{
```

```
int j,t;
for(int i=1;i<N;i++)
{
t=A[i];
j=i-1;
while(A[j]>t)
{
A[j+1]=A[j];
j--;
}
A[j+1]=t;
}
}
```

2. 选择排序

选择排序的基本思想是每步从待排序的记录中选出排序码最小的记录，顺序存放在已排序的记录序列的后面，直到全部排序完成。选择排序中主要使用直接选择排序和堆排序。

代码如下：

```
1. void Choose(int A[],int n)
2. {
3. int k,t;
4. for(int i=0;i<n-1;i++)
5. {
6. k=i;
7. for(int j=i+1;j<n;j++)
8. {
9. if(A[j]<A[k]) k=j;
10.}
11.if(k!=i)
12.{
13.t=A[i];
14.A[i]=A[k];
15.A[k]=t;
16.}
17.}
18.}
```

在选择排序中，如果待排序的数据用高效的优先队列来维护，则插入排序（时

间复杂度为 $O(n^2)$）变为了堆排序（时间复杂度为 $O(n\log n)$）。

```
void Dir_Choose(int A[],int n) //直接选择排序
{
int k,t;
for(int i=0;i<n-1;i++)
{
k=i;
for(int j=i+1;j<n;j++)
{
if(A[j]<A[k]) k=j;
}
if(k!=i)
{
t=A[i];
A[i]=A[k];
A[k]=t;
}
}
}
```

堆排序是一种不稳定的排序，也是一种二叉堆，和二叉查找树有类似之处，是对直接选择排序的有效改进。n 个关键字序列 K_1，K_2，…，K_n 称为堆，当且仅当该序列满足（$K_i \leqslant K_{2i}$ 且 $K_i \leqslant K_{2i+1}$）或（$K_i \geqslant K_{2i}$ 且 $K_i \geqslant K_{2i+1}$）（$1 \leqslant i \leqslant n/2$）。根结点的关键字是堆里所有结点关键字中最小者的堆，称为小根堆；根结点的关键字是堆里所有结点关键字中最大者的堆，称为大根堆。

若将此序列所存储的向量 $R[1..n]$看作是一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于（或不小于）其左右孩子（若存在）结点的关键字。

堆排序的关键步骤有两个：一是如何建立初始堆；二是当堆的根结点与堆的最后一个结点交换后，如何对少了一个结点后的结点序列做调整，使之重新成为堆。

堆排序的精髓在于数据结构。堆排序是就地排序，辅助空间为 $O(1)$。

代码如下：

```
void heapAjust(int array[],int root,int n)
{
//root 指向需要调整的父结点，n 表示数组长度
  int i = root;
```

```
    int j = root*2;
    int tmp;
    while(j <= n)
    {
      if(j < n && array[j+1] < array[j])
          j = j+1;
      if(array[i] > array[j])
      {
          tmp = array[i];
          array[i] = array[j];
          array[j] = tmp;
          int i = j;
          int j = i*2;
      }
      else
      {
          j = n+1;
      }
    }
}
void heapSort(int array[],int n)
{
   int i;
   int tmp;
   for(i = n/2;i > 0;i--)
   {
    heapAjust(array,i,n);
   }
   for(i = n; i > 1;i--)
   {
      tmp = array[i];
      array[i] = array[1];
      array[1] = tmp;
      heapAjust(array,1,i-1);
   }
}
int main()
{
    int i;
    int array[]={0,5,7,6,2,1,4,8};
    heapSort(array,7);
    for(i = 1;i <= 7;i++)
    {
        printf("%4d",array[i]);
```

```
    }
    return 0;
}
```

3. 快速排序

快速排序是一种不稳定的排序。快速排序的精髓在于随机化。快速排序运用了分治法，其基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题，然后递归地解这些子问题，最后将这些子问题的解组合为原问题的解。

快速排序的具体过程如下：

第一步，在待排序的 n 个记录中任取一个记录，以该记录的排序码为准，将所有的记录分成两组，第 1 组中各记录的排序码都小于或等于该排序码，第 2 组中各记录的排序码都大于该排序码，并把该记录排在这两组中间。

第二步，采用同样的方法，对左边的组和右边的组进行排序，直到所有的记录都排到相应的位置为止。

代码如下：

```
void swap(int *a, int *b) //交换函数
{
        int temp = *a;
        *a = *b;
        *b = temp;
}

int partition(int *array, int low, int high)
{
        int middle = (low+high)/2, temp, pivot,i,j;
        //选择第一个元素，最后一个元素，中间元素中的中间值作为支点

        if (array[middle] < array[low])
                swap(&array[middle], &array[low]);
        if (array[high] < array[low])
                swap(&array[high], &array[low]);
        if (array[high] < array[middle])
                swap(&array[high], &array[middle]);

        pivot = array[middle]; // 选中支点

        swap(&array[middle], &array[high-1]);//将支点值换到倒数第二个位置
```

```
        for (i=low, j=high-1; ;) {
                while (array[++i]<pivot) {} //找到一个大于支点的元素
                while (pivot<array[--j]) {} //找到一个小于支点的元素
                if (i < j) { //交换两个元素
                        temp = array[j];
                        array[j]=array[i];
                        array[i]=temp;
                } else
                        break;
        }
        swap(&array[i], &array[high-1]); //将支点换回 i 点，第一次分组结束

        return i;
}
1. void Quick_Sort(int A[],int low,int high) //low 和 high 是数组的下标
2. {
3. if(low<high)
4. {
5. int temp,t=A[low];
6. int l=low,h=high;
7. while(l<h)
8. {
9. while(A[l]<t) l++;
10. while(A[h]>=t) h--;
11. if(h>l)
12. {
13. temp=A[l];
14. A[l]=A[h];
15. A[h]=temp;
16. }
17. }
18. Quick_Sort(A,low,l-1);
19. Quick_Sort(A,l+1,high);
20. }
21. }
void Quick_Sort(int A[],int low,int high) //low 和 high 是数组的下标
{
if(low<high)
{
int temp,t=A[low];
int l=low,h=high;
while(l<h)
{
```

```
while(A[l]<t) l++;
while(A[h]>=t) h--;
if(h>l)
{
temp=A[l];
A[l]=A[h];
A[h]=temp;
}
}
Quick_Sort(A,low,l-1);
Quick_Sort(A,l+1,high);
}
}
```

4. 归并排序

归并排序是一种外排序方法，也是一种稳定的排序，其精髓在于分治法。归并排序将两个或两个以上的有序子表合并成一个新的有序表。初始时，把含有 n 个结点的待排序序列看作由 n 个长度都为 1 的有序子表组成，将它们依次两两归并得到长度为 2 的若干个有序子表，再对它们两两进行合并，直到得到长度为 n 的有序表。

归并排序可用顺序存储结构，也易于在链表上实现，对长度为 n 的文件，需进行 $\log n$ 趟二路归并，每趟归并的时间为 $O(n)$，故其时间复杂度在最好和最坏情况下均是 $O(n\log n)$。归并排序需要一个辅助向量来暂存两个有序子文件归并的结果，故其辅助空间复杂度为 $O(n)$。显然，它不是就地排序。

代码如下：

```
1. void merge(int *array, int *temp_array, int left, int right, int
right_end)
2. {
3.          int left_end = right - 1, i;
4.          int tmp = left;
5.          int num = right_end - left+1;
6.
7.          while (left <= left_end && right <= right_end)
8.                  if (array[left] <= array[right])
9.                          temp_array[tmp++] = array[left++];
10.                 else
11.                         temp_array[tmp++] = array[right++];
12.
13.         while (left <= left_end)
```

```
14.                 temp_array[tmp++] = array[left++];
15.         while (right <= right_end)
16.                 temp_array[tmp++] = array[right++];
17.         for (i=0; i<num; i++, right_end--)
18.                 array[right_end] = temp_array[right_end];
19. }
20.
21. void m_sort(int *array, int *temp_array, int left, int right)
22. {
23.         int center;
24.         if (left < right) {
25.                 center = (left+right)/2;
26.                 m_sort(array, temp_array, left, center);
27.                 m_sort(array, temp_array, center+1, right);
28.                 merge(array, temp_array, left, center+1, right);
29.         }
30. }
31. void merge_sort(int *array, int size)
32. {
33.         int *temp = (int*)malloc(size);
34.         memset(temp, 0, size);
35.         m_sort(array, temp, 0, size-1);
36.         free(temp);
37. }
```

5. 基数排序

基数排序的基本思想是：从低位到高位依次对待排序的关键字（r 个）进行分配和收集，经过 d 趟分配和收集后，就可以得到一个有序序列。

设每个关键字的取值范围均是 $C_0 \leqslant C_j \leqslant C_{rd}$（$0 \leqslant j \leqslant rd$），可能的取值个数 rd 称为基数。基数的选择和关键字的分解因关键字的类型而不同。

若关键字是十进制整数，则可按个、十等位进行分解。

比如：对三个数字（13，21，11）进行排序，则关键字的取值为 $r=0\sim9$，$d=2$，第一次排序为对个位数进行排序，为 21，11，13，第二次排序为对十位数进行排序，为 11，13，21，排序完成。

若关键字是英文字符串，则可按照字符的 ASCII 码进行分解。

基数排序中很常见的一种排序就是桶排序。桶排序的思想是把 [0，1]划分为 r

个大小相同的子区间，每一子区间是一个桶。然后将要排序的 n 个数分布到各个桶中。由于分布是均匀的，所以不会有很多数落在一个桶中的情况。为了得到结果，先对各桶中的元素进行排序，然后按次序把各桶中的元素列出来即可。

各种排序小结如下。

从排序所耗费的计算时间上说，按平均时间将排序可分为如下三种。

- 平方阶（$O(n^2)$）排序：一般称为简单排序，例如，插入排序、选择排序；
- 线性对数阶（$O(n\log n)$）排序：如快速、堆和归并排序；
- 线性阶（$O(n)$）排序：如基数排序。

在不同的环境下，挑选排序的大致原则如下。

1）需要存储在硬盘上的海量数据排序因为数据量大，只能存储在硬盘上，所以应该用归并排序。

2）内存上的排序规则如下。

① 若 n 较小（如 $n \leqslant 50$），排序方法可随意选择；

② 若 n 较大，则应采用时间复杂度为 $O(n\log n)$的排序方法：快速排序、堆排序。

这两种排序的效果最明显。

- 快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
- 堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。

2.2.2 数据结构

基础的数据结构是：数组和链表。数组是排列在内存中连续的对象；而链表是分散存储的多个对象，前后关系用指针表示。目前在工业界主要应用的数据结构有：栈、队列、字典和树。其中，栈是后进先出的数据结构；队列是先进先出的数据结构。

1. 字典

字典与队列和栈不同，它是根据内容读写数据的。Hash（哈希）表是实现字典

的基础数据结构。Hash 表和数组类似，其基本设计思想是：设计一个函数，根据键值直接计算出元素在某数组中的具体位置，而不需要访问其他元素。为了避免过大，我们允许出现冲突，即多个元素被映射到同一个数组位置。解决 Hash 表冲突的方法有开放地址法和链地址法。

开放地址法的基本思想是：当冲突出现后，用当前位置的下一个位置来作为存储数据的位置。只要找到一个够大的，包含很多空洞的 Hash 表，开发地址法在查找数据上还是很高效的。但需要注意的是，开放地址法不支持删除操作。

链地址法的基本思想是：为每个 Hash 表项设置一个链表，新插入的数据直接放在链表中即可。显然，链地址法支持删除操作。

2. 队列

在队列中，比较高科技含量的就是优先队列（第一个元素是最大或者最小元素的队列），优先队列的一个典型实现就是二叉堆。

明白了一般队列也就明白了队列的精髓之处。

队列可以用数组实现，也可以用链表实现。用数组实现时，难点在于：每次删除都需要重新移动之后的所有元素，所以降低了效率。为了避免这一点，在实现时，维护两个下标，分别用于表示第一个元素和最后一个元素。插入和删除操作均不需要涉及元素移动操作，只需要改动相应的下标，就可以不用考虑清除以前使用的内存单元。

如果需要考虑以前使用的内存单元，可以用环形链表的方法。区分队空和队满的方法可以是设置一个变量记录队列中的元素个数。

具体实现如下：

```
typedef struct {
int q[QUEUESIZE+1]; /* 队列主体 */
int first; /* 第一个元素的位置 */
int last; /*最后一个元素的位置*/
int count; /*队列的元素数量*/
} queue;
init_queue(queue *q)
{
q->first = 0;
q->last = QUEUESIZE-1;
```

```
q->count = 0;
}
enqueue(queue *q, int x)
{
if (q->count >= QUEUESIZE)
printf("Warning: queue overflow enqueue x=%d\n",x);
else {
q->last = (q->last+1) % QUEUESIZE;
q->q[ q->last ] = x;
q->count = q->count + 1;
}
}
int dequeue(queue *q)
{
int x;
if (q->count <= 0) printf("Warning: empty queue dequeue.\n");
else {
x = q->q[ q->first ];
q->first = (q->first+1) % QUEUESIZE;
q->count = q->count - 1;
}
return(x);
}
int empty(queue *q)
{
if (q->count <= 0) return (TRUE);
else return (FALSE);
}
```

3. 树

树是最有用的数据结构。常用的树包括：二叉树、红黑树、B-树。比如：stl 中 set 和 map 的底层都是用红黑树实现的；常见的 K-d 树往往是一种二叉树；数据库往往是利用 B-树在硬盘上实现的。

（1）二叉树

在计算机科学中，二叉树是每个结点最多有两个子树的树结构。通常，子树被称作“左子树”（left subtree）和“右子树”（right subtree）。

二叉树的每个结点至多有两棵子树（不存在度大于 2 的结点），二叉树的子树有左右之分，次序不能颠倒。二叉树的第 i 层至多有 2^{i-1} 个结点；深度为 k 的二叉树至多

有 2^k-1 个结点。

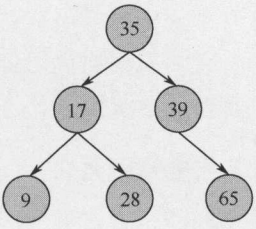


图 2-2　二叉树结构

树的结构如图 2-2 所示。

一棵深度为 k，且有 2^k-1 个结点的树称为满二叉树；深度为 k，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中序号为 1 至 n 的结点对应时，称为完全二叉树。

二叉树常被用于实现二叉查找树和二叉堆。二叉查找树或者二叉堆是一种排序树，每个结点的左子树的结点值总是小于结点本身的值，且每个结点的右子树的值总是大于结点本身的值。

由于图 2-3 所示的链表也属于二叉树的一种，所以，在查找和删除等操作上毫无优势。

由于我们总是希望能利用二分法很快查找到所需的数据，所以我们期望的二叉树是图 2-4 所示的未退化的二叉树。

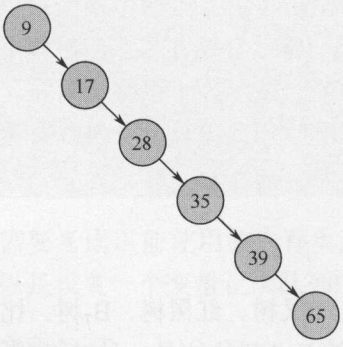


图 2-3　退化的二叉树：链表

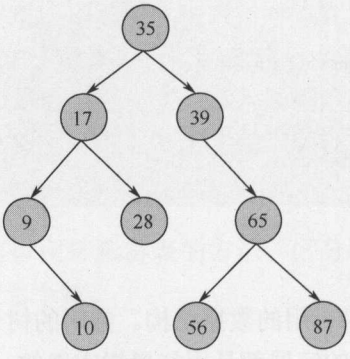


图 2-4　未退化的二叉树

为了避免出现链表形式的退化的二叉树数据结构，我们需要对二叉树进行平衡操作，即“平衡二叉树”，也就是下面将要介绍的红黑树。

（2）红黑树

红黑树是很典型的一种自平衡二叉树，也是非常实用的树，如上文所说，C++ 中的 map 和 set 容器就是用红黑树实现的。

红黑树属于一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是 Red 或 Black。

通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的。

二叉查找树的一般性质如下。

- 在一棵二叉查找树上，执行查找、插入、删除等操作的时间复杂度为 $O(\log n)$。这是因为一棵由 n 个结点随机构造的二叉查找树的高度为 $\log n$；
- 若是一棵具有 n 个结点的线性链，则执行查找、插入、删除等操作的最坏情况运行时间为 $O(n)$。

对红黑树来说，能保证在最坏情况下，基本的动态几何操作的时间均为 $O(\log n)$。

红黑树上每个结点内含五个域：color、key、left、right 和 p。如果相应的指针域没有，则设为 NIL。一般来说，红黑树满足以下性质。

性质 1：每个结点要么是红的，要么是黑的；

性质 2：根结点是黑的；

性质 3：每个叶结点即空结点（NIL），是黑的；

性质 4：如果一个结点是红的，那么它的两个儿子都是黑的；

性质 5：对每个结点而言，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

红黑树的数据结构如图 2-5 所示。

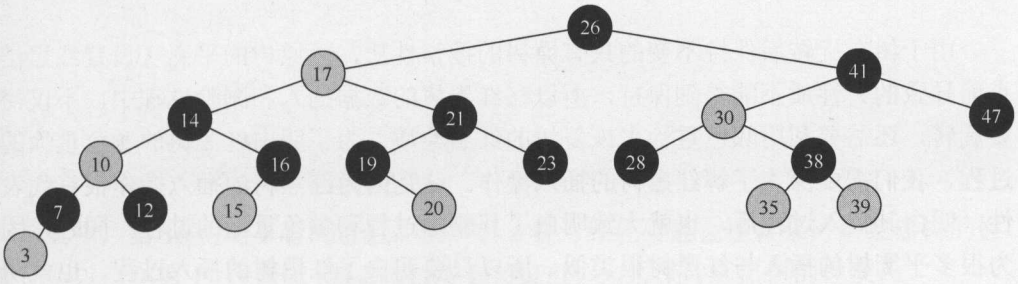


图 2-5　红黑树的数据结构

在对红黑树的结点进行插入、删除等操作时，利用旋转，红黑树依然能保持它特有的性质。

树的旋转分为左旋和右旋。左右旋也是相互对称的，明白一种旋转即可。下面以左旋为例进行介绍，如图 2-6 所示。

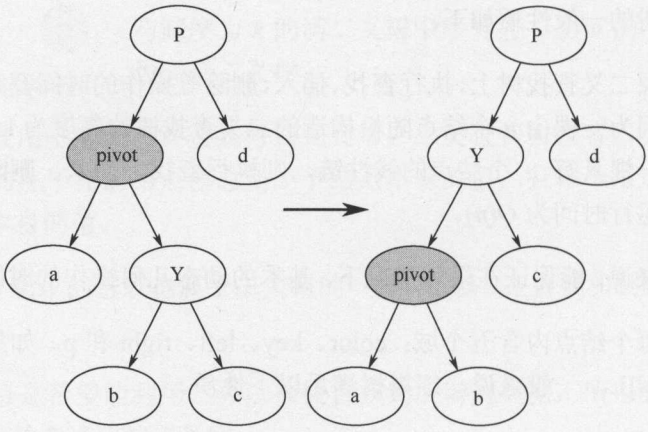


图 2-6　左旋

左旋的原理如下。

当在图 2-7 中某个结点 pivot（pivot 可以为树内任意右孩子，而不是 NIL 的结点）上做左旋操作时，我们得到它的右孩子 Y。

左旋以 pivot 到 Y 之间的链为“支轴”进行，pivot 被逆时针左旋转到 Y 的下侧，从而使 Y 成为该孩子树新的根。为了保持 Y 只有两个孩子，Y 的左孩子 b 则成为（已被旋转为 Y 孩子的）pivot 的右孩子，旋转后形成的树如图 2-6 右侧所示。

同理，树的右旋如图 2-7 所示。

由于树在旋转后保持不变的只有原树的搜索性质，而原树的平衡（设置红黑结点而导致的）性质不能得到保持，所以在红黑树的数据插入和删除过程中，不仅需要旋转，还需要利用颜色重涂来恢复树的红黑性质。为了明白红黑树的颜色重涂的过程，我们需要深入了解红黑树的插入操作。这是因为红黑树的插入操作很有代表性，明白其插入过程后，也就大致明白了其删除过程和颜色重涂的过程。同时，因为很多平衡树的插入与红黑树很类似，所以只要明白了红黑树的插入过程，也就明白平衡树保持平衡的原理。

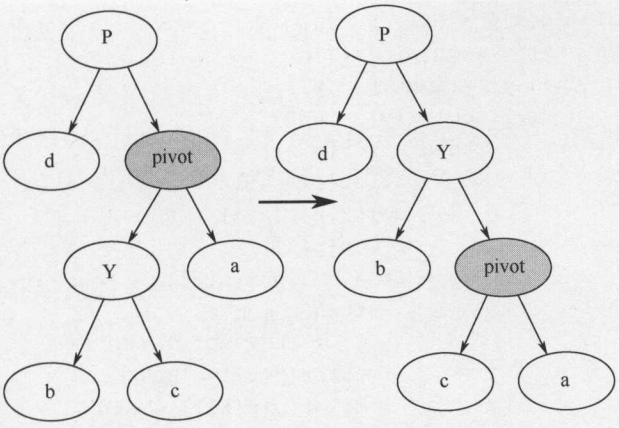


图 2-7　右旋

下面以一个例子来描述红黑树的插入过程。

示例：向一棵含有 n 个结点的红黑树插入一个新结点。

伪代码如下：

```
RB-INSERT(T, z)  //将 z 插入红黑树 T 之内
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4       do y ← x
 5          if key[z] < key[x]
 6             then x ← left[x]
 7             else x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else if key[z] < key[y]
12               then left[y] ← z
13               else right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

其中，第 16 行将 z 着为红色，由于将 z 着为红色可能会违背某一条红黑树的性质，所以，在第 17 行调用 RB-INSERT-FIXUP(T，z)来保持红黑树的性质。

```
RB-INSERT-FIXUP(T, z)
```

```
 1 while color[p[z]] = RED
 2     do if p[z] = left[p[p[z]]]
 3           then y ← right[p[p[z]]]
 4                if color[y] = RED
 5                   then color[p[z]] ← BLACK                    ▷ 情况 1
 6                        color[y] ← BLACK
 7                        color[p[p[z]]] ← RED
 8                        z ← p[p[z]]
 9                   else if z = right[p[z]]
10                           then z ← p[z]                       ▷ 情况 2
11                                LEFT-ROTATE(T, z)
12                        color[p[z]] ← BLACK                    ▷ 情况 3
13                        color[p[p[z]]] ← RED
14                        RIGHT-ROTATE(T, p[p[z]])
15           else (same as then clause
                       with "right" and "left" exchanged)
16 color[root[T]] ← BLACK
```

在对红黑树进行插入操作时，我们一般总是插入红色的结点，因为这样可以在插入过程中尽量避免对树的调整。

插入一个红色结点只可能会破坏红黑树的性质 2、性质 4，其应对策略如下。

其一，把出现违背红黑树性质的结点向上移，如果能移到根结点，那么很容易就能通过直接修改根结点来恢复红黑树的性质。

其二，穷举所有的可能性，之后把能归于同一类方法处理的归为同一类，不能直接处理的归到下面的三种情况。

情况 1：插入的是根结点。

原树是空树，此种情况只会违反性质 2。

应对：直接把此结点涂为黑色。

情况 2：插入的结点的父结点是黑色。

此种情况不会违反性质 2 和性质 4，红黑树没有被破坏。

应对：什么也不做。

情况 3：当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。

此时父结点的父结点一定存在，否则插入前就已不是红黑树。

与此同时，又分为父结点是祖父结点的左子还是右子。对于对称性，我们只要解开一个方向就可以。在此，我们只考虑父结点为祖父左子的情况。同时，还可以分为当前结点是其父结点的左子还是右子，但是其处理方式是一样的，我们将此归为同一类。

应对：将当前结点的父结点和叔叔结点涂黑，祖父结点涂红，把当前结点指向祖父结点，从新的当前结点重新开始算法计算。

（3）B-树

B-树（Balanced Tree）是存储在硬盘上的一种平衡树。之所以要存储在硬盘，是因为 B-树一般是处理大数据量的数据，比如数据库。

总的来说，B-树是磁盘特点与平衡树特点的结合，是一种多路搜索树，其特点如下。

- 定义任意非叶子结点最多只有 M 个儿子，且 M 大于或等于 2；
- 根结点的儿子数为[2，M]；
- 除根结点以外的非叶子结点的儿子数为[M/2，M]；
- 每个结点存放至少 M/2–1（取上整）和至多 M–1 个关键字（至少两个关键字）；
- 非叶子结点的关键字个数等于指向儿子的指针个数–1；
- 非叶子结点的关键字：K[1]，K[2]，…，K[M–1]，且 K[i]<K[i+1]；
- 非叶子结点的指针：P[1]，P[2]，…，P[M]，其中，P[1]指向关键字小于 K[1]的子树，P[M]指向关键字大于 K[M–1]的子树，其他 P[i]指向关键字属于（K[i–1]，K[i]）的子树；
- 所有的叶子结点位于同一层，如 M=3。

体现以上特点的 B-树的结构如图 2-8 所示。

B-树的搜索从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中，则结束，否则进入查询关键字所属范围的儿子结点重复，直到所对应的儿子指针为空，或已经是叶子结点。

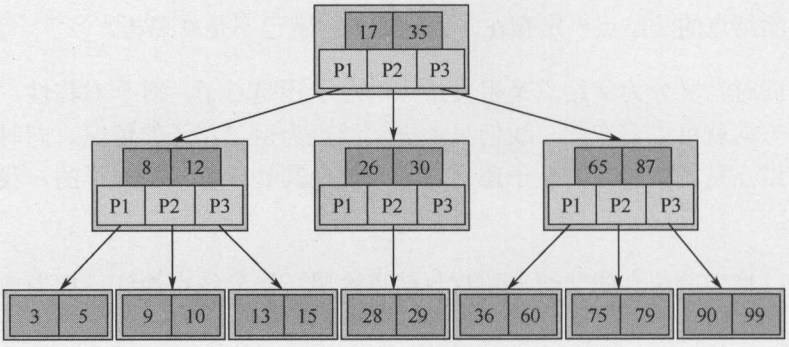


图 2-8　B-树的结构

由于限制了除根结点以外的非叶子结点，至少含有 $M/2$ 个儿子，确保了结点的最少利用率，其最低搜索性能如下。

$$
\begin{aligned}
&O\left[\log_2\left(\left[\frac{M}{2}-1\right]\right)\times\log_{\frac{M}{2}}\left(\frac{N}{\left|\frac{M}{2}-1\right|}\right)\right]\\
&=O\left[\log_2\left(\frac{M}{2}\right)\right]\times O\left(\log_{\frac{M}{2}}\left(\frac{N}{\frac{M}{2}}\right)\right)\\
&=O\left[\log_2\left(\frac{M}{2}\right)\times\left(\log_{\frac{M}{2}}N-1\right)\right]\\
&=O\left[\log_2 N-\log_2\left(\frac{M}{2}\right)\right]\\
&=O[\log_2 N]-O[C]\\
&=O[\log_2 N]
\end{aligned}
$$

其中，M 为设定的非叶子结点最多子树的个数，N 为关键字总数。

从其搜索性能可以看出，B-树的性能总是等价于与 M 值无关的二分查找，也没有二叉树需要平衡的问题。

由于 $M/2$ 的限制，在插入结点时，如果结点已满，则需要将结点分裂为两个各占 $M/2$ 的结点；删除结点时，需将两个不足 $M/2$ 的兄弟结点合并。

（4）B+树

B+树是B-树的变体，也是一种多路搜索树，与B-树相比，不同之处在于如下四点。

- 非叶子结点的子树指针与关键字个数相同；
- 非叶子结点的子树指针 $P[i]$指向关键字值属于$[K[i], K[i+1]]$的子树（B-树是开区间）；
- 为所有的叶子结点增加一个链指针；
- 所有的关键字都在叶子结点出现。

B+树的结构如图2-9所示。

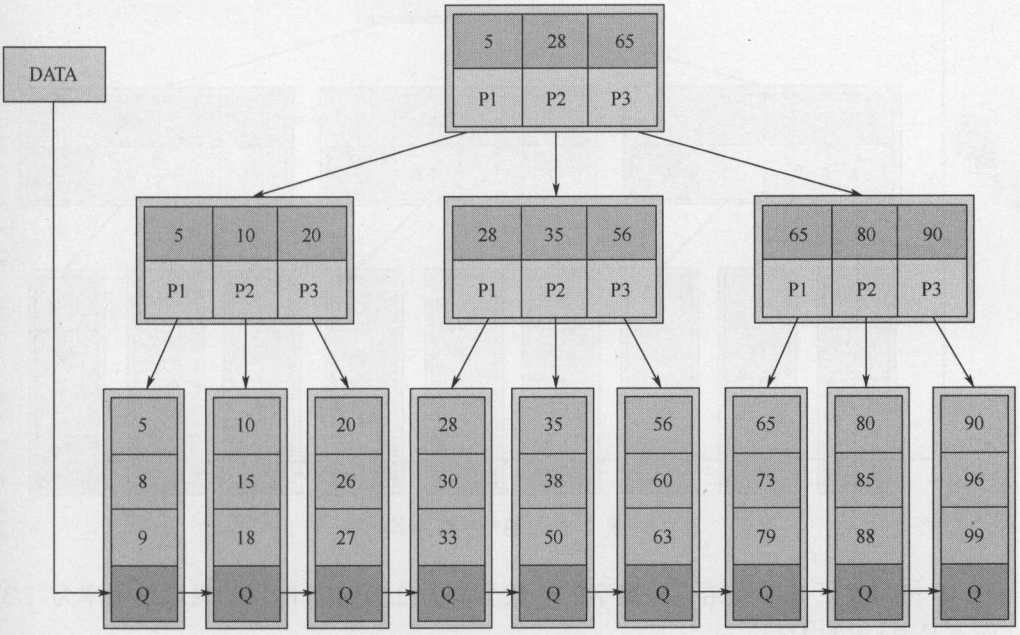


图2-9　B+树的结构

B+树搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中，而B-树则在非叶子结点命中。

B+的特性如下。

- 所有的关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

- 不可能在非叶子结点命中；
- 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层。
- 更适合文件索引系统。

（5）B*树

B*是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针，B* 树的结构如图2-10所示。

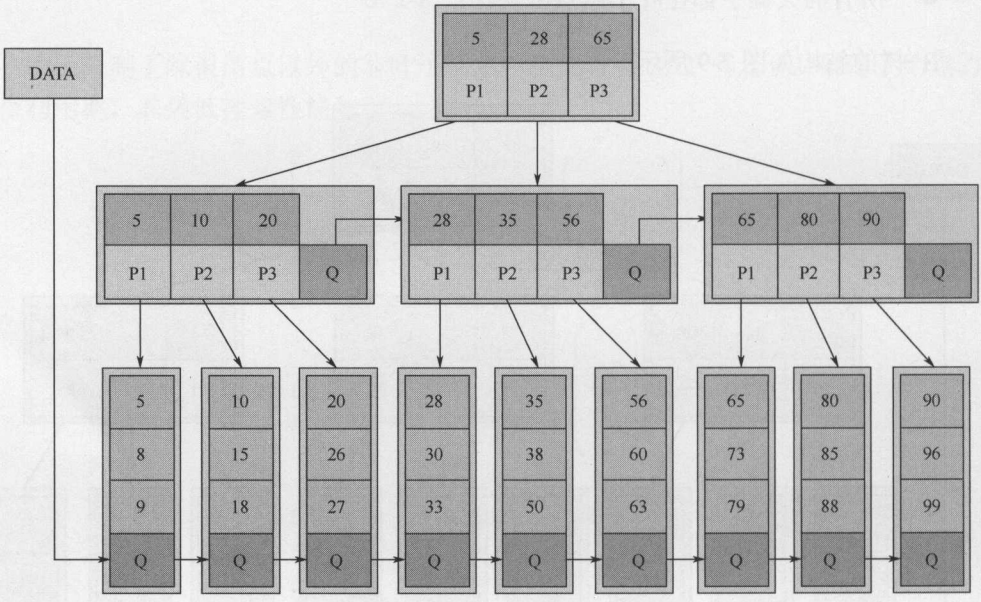


图 2-10　B*树的结构

B*树定义了非叶子结点关键字的个数至少为(2/3)*M，即块的最低使用率为 2/3（而不是B+树的1/2）。

B*树和B+树的一个重要区别就是分裂的过程不同。

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中 1/2 的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以，它不需要指向兄弟的指针。

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分

数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围已改变）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制 1/3 的数据到新结点，最后在父结点增加新结点的指针。

所以，B*树分配新结点的概率比 B+树要低，但是空间使用率更高。

第 2 部分　LBS 常用技术架构

应用程序架构的本质是使应用程序条理化，提高应用程序的性能，并方便拓展和维护。

LBS 应用多种多样，比如：O2O 的应用与社交应用截然不同，但其技术架构有相同之处，都可以分为：数据和引擎两个部分。

第 3 章

LBS 数据及编译的架构

世界正在从 IT 时代步入 DT（数据科技）的时代，一切都是数据。

——马云

3.1 数据的架构

与 LBS 有关的数据多种多样，大致可以分为：用户数据、地图数据和显示文字数据。其中，用户数据由不同的应用来自定义，比如：用户的聊天记录或对战记录；文字数据往往是和兴趣点相关的，比如：商铺信息或用户的微博信息。用户数据和文字数据架构的构建不是 LBS 数据构建的难点，LBS 数据的难点在于对地图数据的理解。

一幅地图数据的大致样子如图 3-1 所示。

图 3-1　地图外观

可以看出，地图数据去除附加的一些属性（如：属性、名称等）。从几何的设计构成上说，主要由点、线、点组成。

- BMD（基本显示模块）：主要是线、面；
- 3D（三维建筑物）：主要是依赖于点；
- POI（兴趣点）：主要是点和文字；
- ROUTE（道路）：主要是线（道路）、点（道路的拓扑点）、线点线（交规）；
- JV（路口指示）：主要是线点线（路口指示）；
- ADMIN（行政区划）：主要是面；
- VOICE（语音）：属于附加属性；
- ORTHO（卫星正交影像）：属于面；
- DTM（数字高程图）：属于面；
- TMC（实时交通信息）：主要是点、线、面三种类型的 TMC。

3.1.1 点

从前面的内容可知，到数据中的点主要包含以下三种。

- 3D（三维建筑物）：主要是依赖于点；
- POI（兴趣点）：主要是点；
- ROUTE（道路）：点（道路的拓扑点）；
- TMC（实时交通信息）：点类型的 TMC。

点的设计可以非常简单，比如：可以只考虑做一个有经纬度的点到地图上就可以，也可以进行更精细的设计，除了建立更好的空间索引来检索外，也可以对点进行排序，以便能更快速地对点进行检索。

对点进行排序的方法主要有两种：希尔伯特曲线排序和 Z 排序。

1. 希尔伯特曲线排序

希尔伯特曲线是一种能填充满一个平面正方形的分形曲线（空间填充曲线），由数学牛人大卫·希尔伯特在 1891 年提出。

由于它能填满平面，它的豪斯多夫维（Hausdorff-Becikovich Dimesion，目前主流的拓扑维度的度量）是 2。

希尔伯特曲线本质上显现了一种排序，这种排序的L系统记法如下。

- 变量：L、R
- 常数：F、+、−
- 规则为：

$$L \rightarrow +RF-LFL-FR+$$

$$R \rightarrow -LF+RFR+FL-$$

其中，F表示向前；−表示右转90°；+表示左转90°。

排序的步骤如图3-2所示。

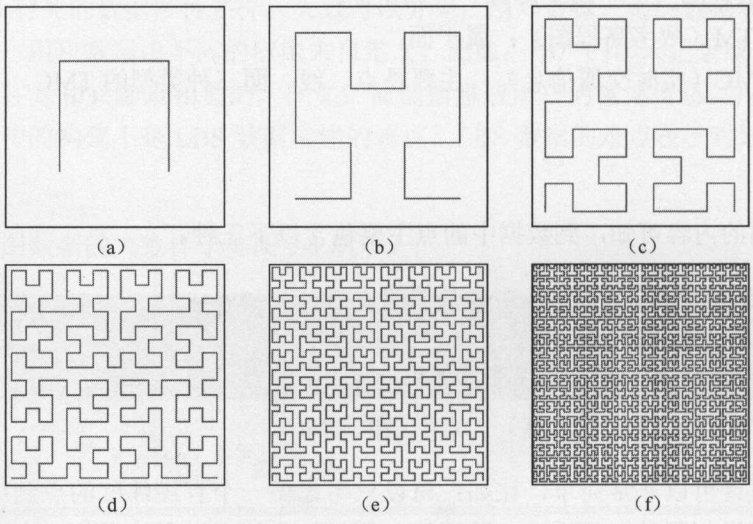


图3-2　希尔伯特排序

2. Z排序（莫顿排序）

与希尔伯特曲线排序一样，Z排序也是一种将多维空间利用曲线进行填满（实际上是一种点排序）的方法。

Z排序是一种很重要的方法，我们将在5.1.2节描述其原理。下面展现一下其排序效果，如图3-3所示。

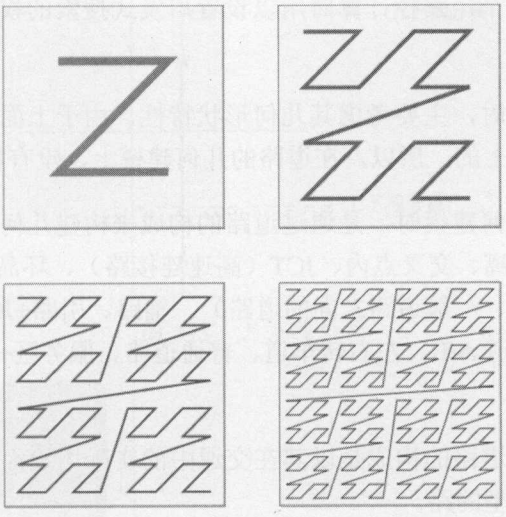


图 3-3　Z 排序

3.1.2　线

道路是由一连串的形状点组成的，如：P1，P2，P3，P4，…，Pn。这从 P1→Pn 所定义的顺序实际上就是导航引擎在渲染形状点时的顺序，即画线方向。

行驶方向是用画线方向来定义的。如果行驶方向为正，意味着行驶方向与画线方向相同，即行驶方向是从画线方向的第一个点到最后一个点。如果行驶方向为负，意味着行驶方向与画线方向相反，即行驶方向是从画线方向的最后一个点到第一个点。

道路的基本特性如下。

- 道路的名称；
- 道路的等级；
- 道路的通行能力等级等。

在这些属性中，最常见的属性是道路的等级和道路的通行能力等级。

道路的等级一般有：高速公路、国道、省道、县道、乡道、县乡村内部道路（村道）、城市快速路、主要道路、次要道路、普通道路、小路。

道路的通行能力等级是一个数字（范围为 0～10），与道路等级有一定的相关性。

其最重要的作用有两个：路径计算时用以设置启发式搜索的权重；保证道路的拓扑连通性。

在进行线的设计时，主要考虑其几何形状特性。由于上面的这些属性往往是附属在道路的几何形状上的，所以，在道路的几何建模上，没有特殊之处。

在进行道路的几何建模时，是通过道路的构成来构建几何道路的。道路的构成具体分为：上下线分离、交叉点内、JCT（高速连接路）、环岛、服务区的道路、引路（高速与一般道路、一般道路之间的道路）、辅路、引路+JCT、出口、入口、右转车道 A/B、左转车道 A/B、左右转车道、普通道路、服务区+引路、服务区+JCT、服务区+引路+JCT。

一般来说，每种道路的构成是以其在交通中的实际用途而定的。但是，在设计中大致分为以两种类型的路。

- 一般路口的道路；
- 复杂路口的道路，是指具备上下线分离特点的道路。

一般路口的道路设计没有什么特别，每条道路用一条几何线来表示。在一般路口中，以最复杂的环岛为例，设计如图 3-4 所示。

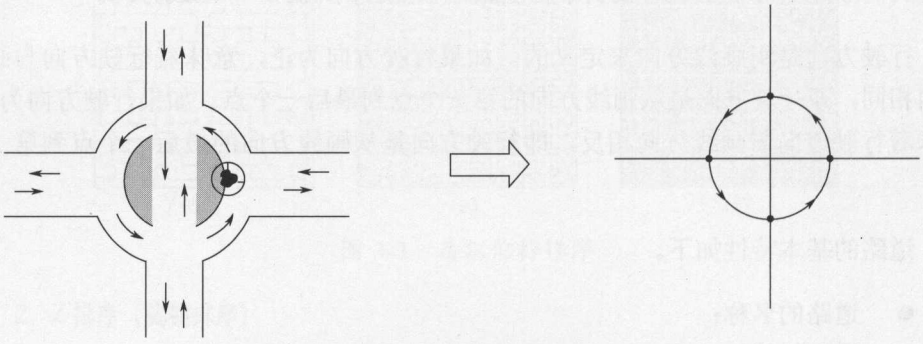


图 3-4　环岛的设计

复杂路口的道路设计往往是和上下线分离的道路相关联的。上下线分离的道路如图 3-5 所示，是指由中央隔离带分隔为上行及下行的两条道路。

上下线分离的道路在路口的地方一般会用交叉点来表示。这种上下线分离的道路的建模过程如图 3-6 所示。

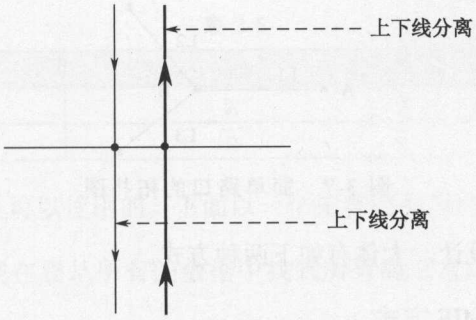


图 3-5　上下线分离

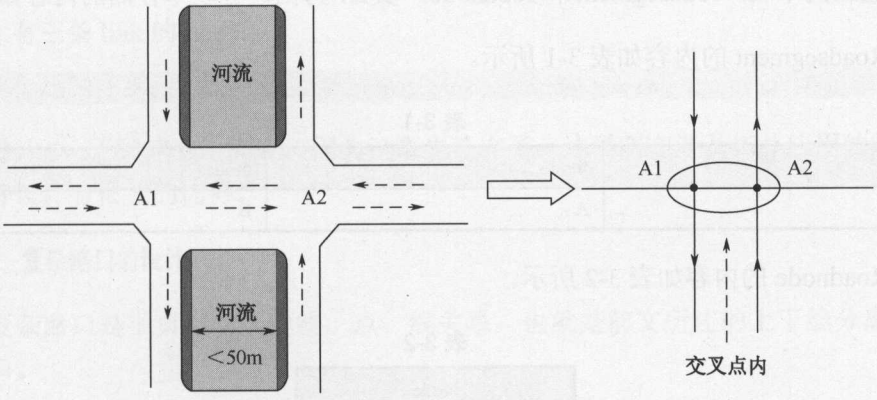


图 3-6　上下线分离的设计

3.1.3　线、点、线

在 LBS 地图中，线、点、线的关系根据路口的类型来分，主要可以分为两种关系：简单路口和复杂路口。

考虑到从某条道路（或道路上的某条车道）可以驶向的道路（或车道）并不相同，所以设定了第三种关系：交规（导航线）。

考虑到道路上与通行有关的标志或警告，所以设定了第四种关系：道路设施或指示。

1. 简单路口的设计

简单路口的关系是一种典型的道路拓扑，其拓扑结构大体如图 3-7 所示。

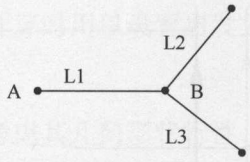


图 3-7　简单路口的拓扑图

关于拓扑结构的设计，大体有如下两种方式。

（1）目前主流的 MIF 方式

建立两个表：roadsegment 和 roadnode。以 L1 为例，各表中存储的内容如下。

Roadsegment 的内容如表 3-1 所示。

表 3-1

Link	fnode	tnode
L1	A	B

Roadnode 的内容如表 3-2 所示。

表 3-2

Node
A
B

（2）以拓扑点为中心建立拓扑关系的方式

建立三个表：roadsegment、roadnode 和 roadnoderelation。以 L1 为例，各表中存储的内容如下。

Roadsegment 的内容如表 3-3 所示；Roadnode 的内容如表 3-4 所示。

表 3-3

Link
L1

表 3-4

Node
A
B

Roadnoderelation 的内容表 3-5 所示。

表 3-5

link	node	nodetype
L1	A	fnode
L1	B	tnode

上述两种方式都是可以使用的。下面以一个拓扑应用为例，来说明两者的区别。

应用：如果我们现在要从所有的数据中找到所有的三岔路口，有以下两种设计方法。

设计 1：需要建立中间表，类似 roadnoderelation。之后再从中找到连接的一个 node，有三条 link 的 node。

```
Select * from roadnoderelation group by node having count() =3;
```

设计 2：无须建立中间表，因为已事先建立了。这说明在涉及拓扑应用的时候，第二种设计有很大的优势。

2. 复杂路口的设计

复杂路口是很典型的一种线、点、线关系，也就是前文所述的上下线分离的道路路口。

复杂路口的拓扑图如图 3-8 所示。

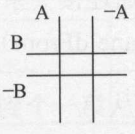


图 3-8　复杂路口拓扑图

复杂路口的拓扑特点是：四个交叉点。这种路口有以下两种设计方法。

（1）目前通用的 MIF 设计方法

将 node 分为两种：普通的 node 和交叉路口的 node。即建立两个表：roadnode 和 roadcross。在这两个表中同时包含了 node 的名称字段。

此外，为了实现导航线（交规），还建立了两种导航线的表：roadnodemaat 和 roadcrossmaat。

主辅路在某种情况下不需要导航线等情况，则需要人为建立的一套规则。

为了能更清晰地表明每条 lane（车道）之间的通行关系，还单独有 lane 的关联表：laneconnectivity。

（2）复杂路口和普通路口的 node 合一的设计

复杂路口和普通路口的 node 合为一种 node，所以只建立一个 node 表：an_nod。

为了区分 node 为交叉路口的情况，或者交叉路口的名称，可在 roadnode 中添加交叉路口的 node 类型和名称，也可成立专门的表：an_ndn(node's name)。

由于 lanecconnectivity 是必须有的，所以将导航线也放在其中，只是一个子集而已。

3. 交规的设计

交通规则即车辆在路网上可以通往的道路（或车道），主要是指一种车道连接的设计。

如果了解了复杂路口的设计，车道连接就容易得多。

（1）一种车道连接的设计

an_lac：道路的连接关系，从进入道路（from chain）到退出道路（to chain）。

an_lat：道路连接内的车道的交通连接关系，包含 lac 中的连接，以及对应的车道间的连接关系，即从进入车道（LaneIdFrom）到退出车道（LaneIdTo）。

an_lch：一个道路连接关系中的从第一个到最后一个道路的序列。

再对比复杂路口图（见图 3-9）来看，整体上看起来还是很清晰的。

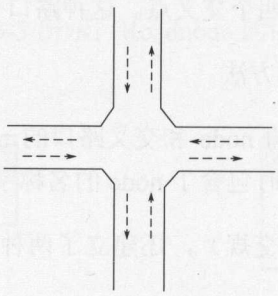


图 3-9　复杂路口

具体设计如下。

an_lac：道路连接表如表 3-6 所示。

表 3-6

列名	类型	外键	说明
Id	INTEGER PRIMARY KEY AUTOINCREMENT		主键
VehiclesImpacted	NUMBER(13)		可以通行的车辆类型
FromChain	NUMBER(10)	AN_CHA.Id (n:0..1)	进入道路
ToChain	NUMBER(10)	AN_CHA.Id (n:0..1)	退出道路

an_lat：道路连接内的车道连接关系如表 3-7 所示。

表 3-7

列名	类型	外键	说明
Id	INTEGER PRIMARY KEY AUTOINCREMENT		主键
ConnectivityId	NUMBER(10)	AN_LAC.Id (n:1)	道路连接关系的 Id
SeqNr	NUMBER(2)		车道连接关系在道路连接内的序号
LaneIdFrom	NUMBER(10)	AN_LAN.Id (n:0..1)	进入车道
LaneIdTo	NUMBER(10)	AN_LAN.Id (n:0..1)	退出车道

an_lch：道路连接中包含的所有道路的序列如表 3-8 所示。

表 3-8

列名	类型	外键	说明
Id	INTEGER PRIMARY KEY AUTOINCREMENT		主键
ConnectivityId	NUMBER(10)	AN_LAC.Id (n:1)	道路连接关系的 Id
ChainId	NUMBER(10)	AN_CHA.Id (n:1)	道路的 Id

注意：an_lch 存储的顺序为从第一个到最后一个道路的序列，比如：一个道路连接的 Id 是 1，对应的 link 是从道路 1 途经道路 2，到达道路 3，则存储如表 3-9 所示。

表 3-9

Id	ConnectivityId	ChainId
1	1	1
2	1	2
3	1	3

如果途经道路有多条，比如：途经道路 2、3，到达道路 4，则在 an_lch 只需要从始发道路到途经道路、到达道路依次存储即可。

为了深入了解这种数据规格的设计，下面对上述设计进行深入分析。

在这种设计中，an_lat 应该包含 node 吗？

在现在的设计中，没有 node，主要是因为复杂路口的情况有四个 node，所以没法定义 lane 的拓扑的 node。这时若要得到复杂路口的四个拓扑点，可以通过 an_lat 和 node 的复杂路口的属性（或由“an_lat 中的 LaneIdFrom 与 LaneIdTo 不连接”来定义复杂路口）编译出复杂路口的四个拓扑点。

在这种设计中，如果交叉路口有虚拟的 lane（如下文第二种设计所述）需要填入到数据中，可以在 an_lat 中添加 node，因为这时“复杂路口的定义”在虚拟车道的威力下已经没有什么意义了，有用的就是 an_lat 中所表达的那种道路、拓扑点、虚拟车道的拓扑关系。

（2）另一种更精细的车道连接设计

从表面上看，这种设计中的关系与上文的设计很不一样。因为其中的很多车道关联关系是建立在 junction（车道连接的结构体）上的。此外，junction 中还有虚拟 link。

具体的精细车道的设计图如图 3-10 所示。

这个设计图实际上与复杂路口的第一种设计所表达的一样，都是复杂路口的情况，但是图 3-10 更精确，因为其将 road 10 与 road 70 相连接的关系中加入了虚拟 link：road 30。同理，road 20 、road 40 也都属于这一类，即为了使车道连接表达更精确而

设定的 link。

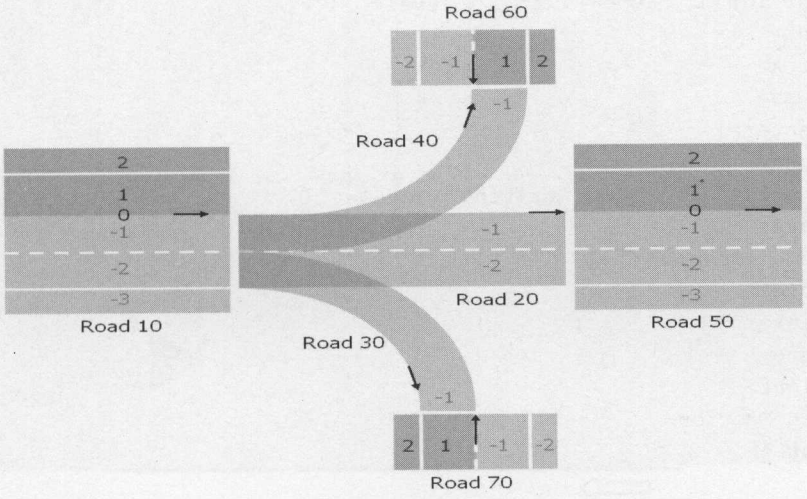


图 3-10　精细车道的设计

实际上，这种车道连接的关系可以叫车道连接，也可以叫 junction。所以这种 junction 没有什么神秘之处，只是将所有的车道连接关系建立了一个 junction 来存储而已。为了使存储更精确，又把一个车道的连接关系分成了三段 lane：出发、目标和中间（这是虚构的 lane）。

上面描述了关于虚拟的 road 与虚拟的 junction 的关系。在这个虚拟的 junction 中，虚拟的 road 的前后继关系是写上相应的 road 的。但是，这不是 junction 的全部含义。

实际上，在这种设计中，利用复杂路口生成了四个新的 junction，也就是上文所述这种设计的实现（为了更好地说明数据结构间的关系，用 XML 来表示具体的车道设计）如下：

```
    <road name=" " length=" 2.4788652606678024e+01 " id=" 13 " junction=" 
27 " >
    <link>
    <predecessor elementType=" road " elementId=" 10 " contactPoint=" end " />
    <successor elementType=" road " elementId=" 70 " contactPoint=" end " />
    </link>
    <lanes>
    <laneSection s=" 0.0000000000000000e+00 " >
    <left>
```

```
</left>
<center>
<lane id="0" type="border" level="0">
<link>
</link>
</lane>
</center>
<right>
<lane id="-1" type="driving" level="0">
<link>
<predecessor id="-2" />
<successor id="1" />
</link>
</lane>
</right>
</laneSection>
</lanes>
```

另外，对应原先已有的四条道路（如 road 10）的情况，其实现如下：

```
<road name="" length="2.0000000000001661e+00" id="10" junction="-1">
<link>
<predecessor elementType="road" elementId="99" contactPoint="end" />
<successor elementType="junction" elementId="25" />
</link>
```

这里注意以下两个细节：

- road junction="-1"
- successor elementType="junction"elementId="25"

4. 道路上的引导指示或设施

道路设施包含：红绿灯、限速牌、各种警告、天桥、隧道等。道路上的指示特指交叉路口的道路指示。

道路设施如图 3-11 所示，图中的 A、B 即为道路设施，道路设施的存储方法一般为与相应的 link 关联，并存储设施本身的经纬度。

只有红绿灯与众不同，在一般的导航地图数据中，由于并不是 ADAS（自动驾驶）地图，所以对精度的要求相对较低，会把红绿灯当作一个路口拓扑点的属性来存储。

道路指示（这里指一般路口中三岔路口的实景路口指示图）如图 3-12 所示。

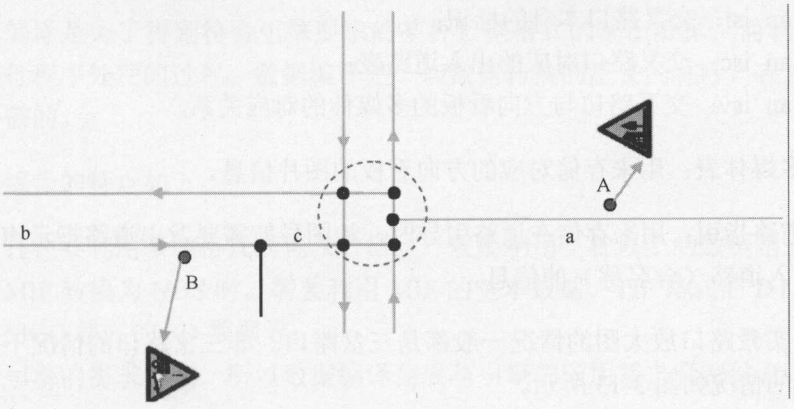


图 3-11　道路设施

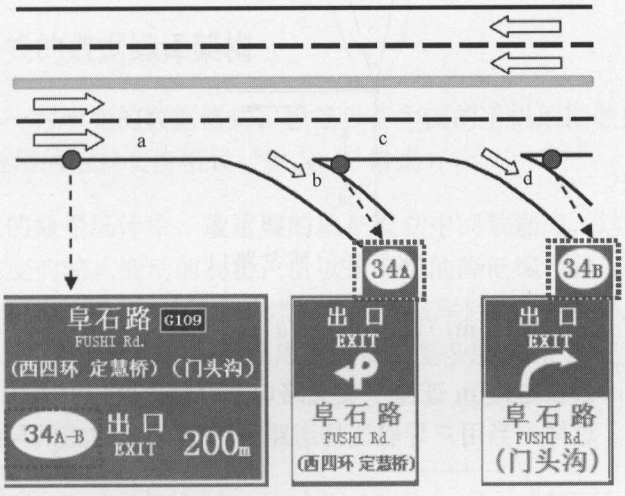


图 3-12　路口指示

在实际的导航地图中，图 3-12 中的 34A-B 并不会实际存储，是由导航软件的判断来实现的，比如：在高速三岔路口前 200m 发出提示。

34A、34B 则会实际存储，具体的存储方法用一个示例来表示。

- 建立三个表，用来表示交叉路口的拓扑关系；
- 建立一个多媒体表，用来存储方向看板的多媒体信息；
- 建立一个表，用来存储道路的指引信息。

1）交叉路口的拓扑关系如下。

- an_isi：交叉路口本身的标识；
- an_isc：交叉路口对应的出入道路表；
- an_isv：交叉路口与方向看板的多媒体的对应关系。

2）多媒体表：用来存储对应的方向看板的图片信息。

3）道路指引：用来存储在道路引导时，地图导航需要发出道路指示声音时所需的对应出入道路（含名称）的信息。

需要实景路口放大图的情况一般都是三岔路口。非三岔路口的情况不太常见，这种路口的情况如图 3-13 所示。

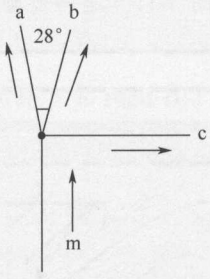


图 3-13　四岔路口

图 3-13 中，进入道路：m；退出道路：a、b、c。

一般情况下，会在道路 m 或整个交叉路口上存储“路口背景图”，会在退出道路上存储箭头图。这样，当用户导航选择道路 m 和 a 后，对应的导航图像如图 3-14 所示。

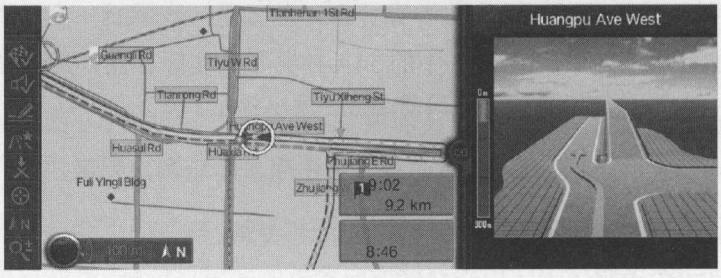


图 3-14　导航图

3.2 数据编译器的架构

数据编译是为了得到符合引擎要求的某种数据格式的输出数据，而利用多种输入数据进行程序处理的过程。数据编译是一切数据转换的泛义的称呼，在 LBS 引擎中也是必需的。

数据编译的特点如下。

- 往往要利用多种格式的输入数据。一般要利用三种以上的数据格式，比如：MIF 转换为 NDS 时，需要利用 MIF 的基本数据、TIF 格式的 DTM 数据、MQO 格式的 3D 数据等；
- 引擎的需求多变，所以数据编译需要与引擎的应用需求紧密结合。

总之，数据编译的需求一般来自两方面：应用的效率需求和甲方的产品需求。

3.2.1 交换格式的数据编译架构

交换格式是一种明文的数据格式，也是目前主流的手机地图数据格式（如 MIF 格式）或 LBS 应用的信息交换格式（如 XML 格式）。

在交换格式的数据编译中，最重要的就是设立中间数据库，这是因为由于适配器是处理复杂多变的输入变动的利器，所以最简单的降低编译程序复杂度的方法就是建立中间数据库，如图 3-15 所示，建立中间数据库后，三个输入到三个输出的复杂度由 9 变为 6。可见，采用中间数据库使编译的复杂度由 $N*N$ 变为 $2N$。

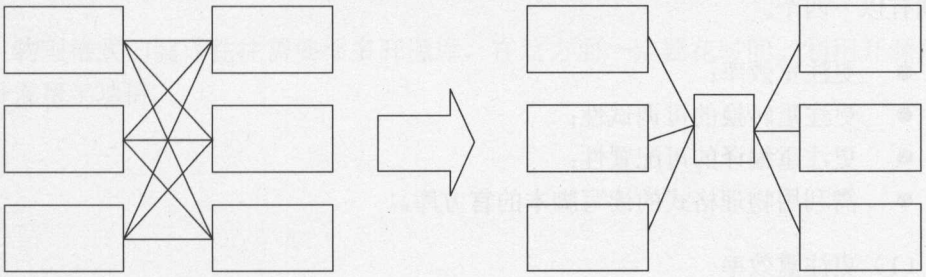


图 3-15　中间数据库

此外，由于数据编译涉及的算法有很多，往往需要用到一些开源库，比如：GDAL、GEOS、Spatial、Magick、PCL 等，为了更方便地对接这些开源库，编译环境建议使用 Linux。

Linux 下三种常见的编译开发技巧如下。

（1）多语言的合作

合作的关键是利用 Shell 语言的脚本作为胶水，去调用其他程序，如：C/C++生成的 exe；python 或 perl 的脚本。

（2）调用配置文件的方法

在 Shell 的脚本中配置编译器的参数（如编译器的某项功能的实现与否）。在 C/C++等语言中调用专门的配置文件（如 poi 配置文件或 bmd 配置文件）。

（3）C 与 Linux 环境变量贯通的方法

格式如下：

Getenv("环境变量名称");

3.2.2 物理格式的数据编译架构

物理格式的数据是非明文的，是 BLOB（二进制字节流）形式的数据，比如：各种结构体或类形式的数据。在调用物理格式的数据时，用调用偏移量来代替交换格式（比如：XML）的解析过程。所以，读取物理格式的数据的速度很快，也有利于引擎在将数据从内存转入硬盘，或者从硬盘转入内存时，不必重新花时间来组织。

物理格式的数据编译与交换格式的数据编译类似，但是技术性会更强，主要的区别有以下四个。

- 更注重效率；
- 更注重转换的可调试性；
- 更注重编译的可配置性；
- 需利用物理格式的读写脚本的官方库。

（1）更注重效率

提高效率的最好方法就是对所有的计算建立好的算法。具体地说，就是要保证建立好的数据结构和对数据结构的计算方法。

有以下两种常用的技巧可保证做到这一点。

在涉及计算时，一定要首先考虑计算的时间复杂度，从而设计合适的算法。

对所有的模块和函数功能都要对耗费时间进行检测（一般输出到日志中），从而保证目前所有的模块以及以后修改升级的模块的运行时间都可控。

（2）更注重转换的可调试性

由于一份完整的中国数据的编译往往要耗费一天甚至更久的时间。所以，为了保证每一步转换过程的中间数据都是可记录的，从而在出现 Bug 时，可以快速找出问题所在。

除记录数据外，基于测试的架构也是快速开发的关键所在。对每一个数据的转换函数，如果得到的数据结构不合预期，则将其输出在日志中。这样在每一次的编译中无须调试，即可第一时间知道数据转换的 Bug 所在。

（3）更注重编译的可配置性

由于不同的甲方需求往往不同，所以为了增加编译器的适应性和拓展性，需要使编译器成为可配置的编译器，即对每一个功能模块都可以用配置文件来选择是否使用这个功能，或者如何使用某个功能（比如面化简的参数），把这些所有的参数用一个配置文件来管理。

（4）需利用物理格式的读写脚本的官方库

对于物理格式的数据，官方往往会提供一些读写物理格式脚本的程序库。正确快速地使用这些库可以加快数据编译的速度，后期有利于数据的可维护性。

物理格式的编译往往需要很多开源库，在这方面一定要花时间。利用开源库能让开发事半功倍。

第 4 章

LBS 引擎的架构

技术含量高的应用程序的架构与 UNIX 操作系统类似；技术含量低的应用程序的架构与 UNIX 操作系统的 UI 类似。

4.1 内存和磁盘

内存和磁盘的运行原理是很多设计模式或数据结构的基础。比如：程序要想运行快，要存储在内存中，而不是硬盘中；数组在按照下标来查找时速度最快。

1. 内存

内存的原理图如图 4-1 所示。

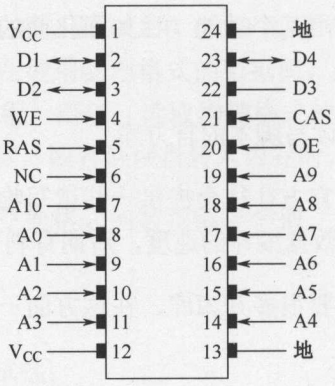


图 4-1

内存的主要特点是对内存里的空间寻址时间是恒定的，速度极快，大概是几个时钟周期。由于 CPU 的计算速度也是以 GHz 来论的，所以，如果读、写或者计算都是在内存中，则处理速度会极快。这种情况是我们期望出现的情况，即程序无须对数据进行遍历查找，而是直接用指针在内存中定位寻址，进而处理。

基础知识章节中作为字典结构基础的 Hash 表（见 2.2.2 节）的原理就是依赖于内存的定位寻址原理而实现的。

2. 外存储器——磁盘

计算机存储设备一般分为两种：内存和外存。内存存取速度快，但容量小、价格昂贵，而且不能长期保存数据（在不通电的情况下，数据会消失）。

磁盘是一种直接存取的存储设备，又称为硬盘。它是以存取时间变化不大为特征的，可以直接存取任何字符组，且容量大，速度较内存而言要慢很多。

现在在新科技的引领下，出现了很多磁盘的改进种类，比如固态硬盘，其查询效率要比一般的硬盘要高很多。由于各种硬盘的作用原理有一定的相似性，所以，如果我们要想明白硬盘的工作原理，只需明白一般硬盘的工作原理，就可以指导 LBS 领域的各种应用开发了。

（1）磁盘的构造

与电唱机的唱片类似，磁盘是一个扁平的圆盘，盘面上有许多称为磁道的圆圈，数据就记录在这些磁道上。磁盘可以是单片的，也可以是由若干盘片组成的盘组，每一个盘片上有两个面。以图 4-2 所示的 6 片盘组为例，除去顶端和底端的外侧面不存储数据之外，一共有 10 个面可以用来保存信息。

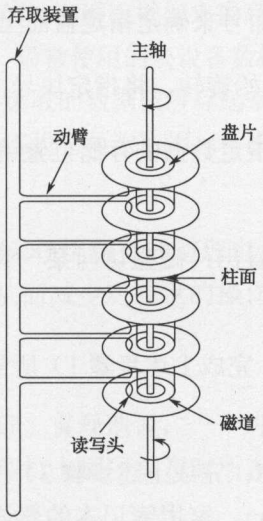


图 4-2　磁盘的构造

当磁盘驱动器执行读/写功能时，盘片装在一个主轴上，并绕主轴高速旋转，当磁道在读/写头（又叫磁头）下通过时，就可以进行数据的读/写。

磁盘一般分为固定头盘和活动头盘。固定头盘的每一个磁道上都有独立的磁头，磁头是固定不动的，专门负责该磁道上数据的读/写。

活动头盘（见图 4-2）的磁头是可移动的，每一个盘面上只有一个磁头，磁头是双向的，因此正反盘面都能读写。它可以从该面的一个磁道移动到另一个磁道。所有的磁头都装在同一个动臂上，因此，不同盘面上的所有磁头都是同时整齐移动的。当盘片绕主轴旋转的时候，磁头与旋转的盘片形成一个圆柱体。各盘面上半径相同的磁道组成了一个圆柱面，这个圆柱面通常简称为柱面。因此，柱面的个数也就是盘面上的磁道数。

（2）磁盘的读/写原理和效率

磁盘中的数据必须用一个三维地址唯一表示：柱面号、盘面号和块号（磁道上的盘块）。

读/写磁盘中某一指定的数据时需要以下三个步骤。

1）动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找。

2）图 4-2 的 6 盘组示意图中，所有的磁头都定位到 10 个盘面的 10 条磁道上（磁头都是双向的），这时根据盘面号来确定指定盘面上的磁道。

3）盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上述步骤的操作后，指定数据的存储位置就被找到了，这时就可以开始读/写操作。

通过上面的三个步骤可以看出：磁盘访问某一具体信息所花费的时间由以下三部分组成。

查找时间（seek time）Ts：完成上述步骤 1）所需要的时间。这部分时间代价最高，最大可长到 0.1s 左右；

等待时间（latency time）Tl：完成上述步骤 3）所需要的时间。由于盘片绕主轴旋转速度很快，一般为 7200r/m。家用笔记本的普通硬盘的转速一般有 5400r/m、7200r/m 几种。因此，一般情况下，旋转一圈大约为 0.0083s；

传输时间（transmission time）Tt：数据通过系统总线传送到内存的时间，一般传输一字节（Byte）大概为 0.02μs（即 2×10^{-8}s）。

磁盘读取数据是以盘块为基本单位的，位于同一盘块中的所有数据都能被一次性全部读取出来。而磁盘 I/O 代价主要花费在查找时间 Ts 上。因此，我们应该尽量将相关信息存放在同一盘块、同一磁道中，或者至少放在同一柱面或相邻柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间 Ts。

所以，在大规模数据存储方面，大量的数据存储在外存磁盘中，而在外存磁盘中读取/写入块（block）中的某数据时，需要定位到磁盘中的某块磁道段才能读写。

那么，如何有效地查找磁盘中的数据呢？这需要一种合理高效的外存数据结构，这就是在基础知识章节的树结构中所介绍的 B-树结构，以及 B-树的变种结构：B+树结构和 B*树结构（见 2.2.2 节）。

4.2 操作系统原理

操作系统中有三个重要的概念：缓存、进程和进程通信。本节将介绍缓存和进程，进程通信在网络传输章节的进程通信部分（即 10.1 节）中介绍。

1. 缓存

UNIX 运用一个功能广泛的缓冲和缓存框架来提高系统的速度。缓冲和缓存利用一部分系统物理内存，确保最重要、最常使用的块设备数据在操作时可直接从主内存获取，无须从低速设备读取。从快设备读取的数据被缓存起来，使得随后对该数据的访问可直接在物理内存进行，而无须从外部设备再次取用。考虑系统中的多种因素，然后延迟写回在总体上改进了系统的性能。

在实际的 LBS 应用中，需要使用缓存的道理也是一样的：在数据库操作和应用的逻辑操作间建立内存缓存，从而减少对硬盘的操作。

2. 进程

进程是操作系统结构的基础。直观地说，一个进程等价于一个正在执行的程序。进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

进程本质上是一个程序的所有代码的汇编语言版本的集合体。所以，进程包含了对寄存器和数据的控制。由于内存的限制，进程只会加载一部分进入内存，其余部分先存储在硬盘的一个特定缓冲区中，等进程运行到适当的位置时，再加载它。

进程对内存的分配是通过 malloc 来实现的。

知道了 malloc，也就知道了进程对计算机资源（内存、文件）的管理，以及对操作系统的大概了解。

malloc 是通过 block 链结构、寻找合适的 block、分裂 block 来实现的。

（1）堆内存空间组织的 block 链结构

在操作系统或 LBS 应用中，对于堆内存空间的组织，一个简单可行的方案通常是将堆内存空间（内存中用来分配给文件的空间）以 block 链的形式组织起来，每个块由 meta 区和数据区组成，meta 区记录数据块的元信息（数据区大小、空闲标志位、指针等），数据区是真实分配的内存区域，并且数据区的第一字节地址即可作为 malloc 返回的地址。

可以用如下结构体定义一个 block（块）：

```
1 typedef struct s_block *t_block;
2 struct s_block {
3     size_t size;  /* 数据区大小 */
4     t_block next; /* 指向下一个块的指针 */
5     int free;     /* 是否是空闲块 */
6     char data[1]  /* 这是一个虚拟字段，表示数据块的第一字节，长度不应计入 meta */
7 };
8
```

堆内存空间的组织如图 4-3 所示。

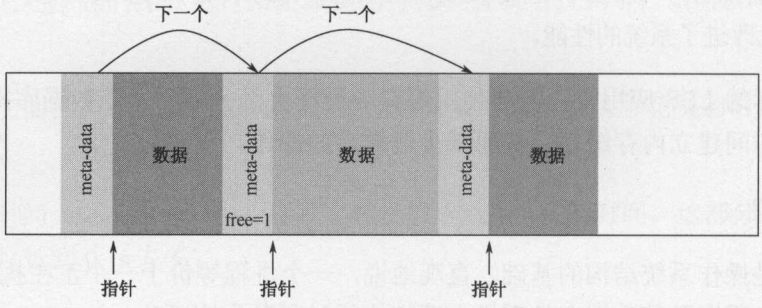


图 4-3　堆内存空间的 block 链组织

（2）寻找合适的 block

如何在 block 链中查找合适的 block？一般来说有以下两种查找算法。

First fit：从头开始，使用第一个数据区的大小大于要求 size 的块作为此次需要

分配的块。

Best fit：从头开始，遍历所有的块，使用数据区的大小大于 size 且差值最小的块作为此次分配的块。

两种方法各有千秋，Best fit 具有较高的内存使用率，而 first fit 在实际中具有更好的运行效率，是运用最广泛的寻找 block 的实用方案。

First fit 算法的代码如下：

```
t_block find_block(t_block *last, size_t size,size_t e) {
    t_block b = first_block;
    while(b && !(b->free && b->size - e >= size)) {
        *last = b;
        b = b->next;
    }
    return b;
}
```

find_block 从 first_block 开始，查找第一个符合要求的 block 并返回 block 起始地址，如果找不到，则返回 NULL。

（3）分裂 block

当 First fit 发现一个较大的空闲块后，会在剩余数据区足够大的情况下将其分裂为一个新的 block，分裂的过程如图 4-4 所示。

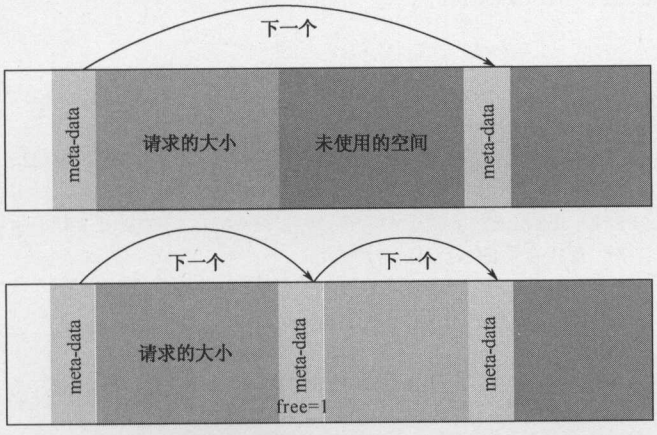


图 4-4　block 的分裂

实现代码如下：

```
1  void split_block(t_block b, size_t s) {
2      t_block new;
3      new = b->data + s;
4      new->size = b->size - s - BLOCK_SIZE ;
5      new->next = b->next;
6      new->free = 1;
7      b->size = s;
8      b->next = new;
9  }
```

需要注意的是，为了防止在分配多次后空闲块中可能会有很多容量极小以致无法分配的内存块。这不仅会浪费内存，还会降低系统寻找空闲块的效率。为了避免这种情况，选定一个常量 s，如果满足分配条件的空闲块的容量为 m，要分配的容量为 n，而 $m-n \leqslant s$，则将 m 整个分配给用户。如果 $m-n > s$，则将 m 分裂。

此外，为了防止每次分配内存都从链表头开始，势必造成存储量小的结点密集在链表头的附近，这同样会增加查询较大空闲块的时间。为了避免这一点，在遍历时会更新一个叫 last 的指针，这个指针始终指向当前遍历的 block，而在下一次查找时，会从 last 开始向后找。

（4）malloc 的实现

根据上面的内容，malloc 的实现代码如下：

```
1  #define BLOCK_SIZE 24
2  void *first_block=NULL;
3
4
5  void *malloc(size_t size) {
6      t_block b, last;
7  size_t s = size;
8
9      if(first_block) {
10         /* 查找合适的block */
11         last = first_block;
12         b = find_block(&last, s,e);
13         if(b) {
14             /* 为了防止碎片而使用的常数。满足这种条件则分裂 */
15             if ((b->size - s) >= ( BLOCK_SIZE + 8))
16                 split_block(b, s);
```

```
17              b->free = 0;
18          } else {
19              /* 没有合适的block，返回空指针 */
20              return NULL;
21          }
22      } else {
23
24              return NULL;
25      }
26      return b->data;
27  }
28
29
```

4.3 设计模式

目前设计模式非常流行。现在只要用搜索引擎搜索，就会搜到很多关于设计模式的资料，但很多内容都让人一头雾水，难以理解。所以在本书中，笔者希望能用一种简明的语言来描述设计模式。

从根本上说，设计模式是程序员的一种心得笔记，是记录各个结构体的关系构造的心得。

我们只从静态和动态（行为）的角度考察 1～3 个结构体的关系，把这种关系叫作设计模式，并把这种关系作为一切结构体间关系的一种抽象。下面将详细介绍它们。

（1）一个结构体

一个结构体是单例模式。

（2）两个结构体 A 和 B

A 和 B 的关系有以下两种。

- 父子关系：工厂模式、抽象工厂；
- 子父关系：装饰模式。

（3）三个结构体 A、B 和 C

A、B、C 的关系如下。

1）从组成角度上看：

- 其中一个是其余两个共有的部分：享元模式；
- 其中一个是其余两个的连接：适配器模式、桥接模式；
- 其中一个是另一个的代表：代理模式；
- 其中一个是其余两个的父亲：组合模式。

2）从行为角度上看：

- A→B→C：职责链模式；
- 其中一个是其余两个行为的观察者：观察者模式；
- 其中一个是其余两个行为的处理者：中介者模式；
- 其中一个是其余两个的父亲：状态模式和策略模式。

4.4 引擎架构

4.4.1 五个要点

对于一个 LBS 引擎，如果能够很快得到用户所需的数据，而且方便维护，那么从技术的角度说，可以认为这是一个好的 LBS 引擎。

由于每一个 LBS 应用都有各自的特点，所以本书只从技术的角度来考虑架构。

设计模式中更多地考虑一个程序中某个模块的安排和设计，而我们需要的是对一个程序从硬件和网络通信的角度出发的总设计。

若要进行一个总体设计，很快得到用户所需的数据，需要以下五种重要的技术。

（1）从硬盘的角度来说

数据库技术：要建立索引；

文件管理系统：建立一个树状的管理系统。

（2）从内存的角度来说

缓存：采用加载一次后即进行内存缓存，从而尽量操作内存，而不是硬盘。

（3）从多进程的角度来说

进程间的通信：尽量用速度最快的共享内存（sharedmemory）。

（4）从网络通信或并发处理的角度来说

利用线程池：可用开源的I/O完成端口（即IOCP，适用于Windows系统）或epoll（适用于Linux系统）。

采用预加载的技术：预先预测用户要加载的内容，在网络服务器端或本地客户端对用户数据进行预加载，从而给用户造成网络通信速度/引擎反应速度很快的感觉。

尽量操作本地数据（比如：优选本地内存缓存，其次本地硬盘缓存已下载的数据等），而少利用网络通信。

（5）从数据转换的角度来说

LBS引擎本质上也是一种对数据的处理过程，所以，在数据编译架构中所提到的效率、可配置、基于测试的架构、开源库在LBS引擎的设计中都是需要的。

在设计LBS引擎时，如果能充分运用这五种技术，则能将LBS引擎设计提升一个档次。

4.4.2 一个失败的案例

为了说明如何进行好的设计，我们先来分析一个失败的设计，这有助于理解一个好的LBS引擎的设计。

2009年，一家小公司在校内网设计了一个App，属于LBS的在线社交应用，其设计示例如下。

1）UI：采用的是FLEX技术；调用动画时，用FLEX调用Flash。

2）服务器引擎设计为三层：数据库、数据操作和用户的操作逻辑。采用的语言为C#。

3）用户通信：在Windows下利用FLEX来调用未经验证的Java的开源Socket程序，每个通信建立一个通信文件，之后在Flash的通信窗口呈现。

4）App的后台服务：由于校内只提供一个网页的Frame接口，当时还没有云服

务，所以，即使开发离线式游戏，也需要开发者购买服务器。

5）引擎的机制：所有的用户数据存储在数据库，每一次的用户操作都会由操作逻辑层调用数据操作层，之后调用数据库，将数据库中得到的结果返回 FLEX 显示。

从表面上看，这个设计很清晰，但存在以下不足。

- FLEX 显然是不主流的技术，未经大规模地验证，速度极慢，体验极不流畅。如果是在电脑上，而且开发者对 C 语言不是很懂，应该用 Flash 技术本身进行设计。
- 没有用缓存或共享内存，所以每次操作都会调用数据库，这是一种硬盘操作，并发性很差。
- 没有考虑分布式的设计，如负载均衡服务器来做负载拓展。
- 网络通信时，采用的开源 Socket 程序的每次通信都建立一个线程，所以反应很慢，而且不能适应高并发的情况。另外，由于对代码不了解，所以无法控制聊天记录。
- 在语言上应该优先考虑速度更快、兼容性更好的 C 或 Java。
- 由于整体的硬件成本（服务器、网址、带宽）和人力成本（对新的技术以及人的学习和试错成本）都很高，所以转换成本很高。第一次不成功，就没有第二次机会了。即使第一次的失败已经给开发人员带来了很多经验，但由于第一次所耗费的成本太高，已经耗去了所有的创业资本。整件事只能就此结束。

所以，整个设计采用的技术都是未经验证的技术，而且难以拓展。

4.4.3 建议

总的来说，在进行架构设计时，应该考虑如下几点。

- 如果财力有限，尽量考虑只做离线式的 App，挂在苹果系统或安卓系统的应用商店就可以；
- 如果一定要做在线式的 App，且小有财力，可以考虑购买云服务（如阿里云）；
- 如果一定要做在线式的 App，可尽量考虑用户的数据尽量少地上传，多用共享内存，即用户只读服务器的数据，而不修改更新服务器的数据。这样就减少了用户对服务器硬盘操作过多的可能性；

- 如果一定要做在线式的 App，且用户数据需要更新服务器数据的操作有很多，财力也足够，就一定要考虑以下设计。

  分布式设计（有一个对用户需求的管理的入口服务器）；

  缓存（客户端和服务器端）；

  开源线程池（IOCP 或 epoll）。

4.4.4 一个 LBS 引擎的实施案例

导航引擎是很典型的一个 LBS 应用，所以这里以一个导航引擎的实施案例为例来说明 LBS 引擎的设计原理。

从业务逻辑上说，总体的架构如图 4-5 所示。

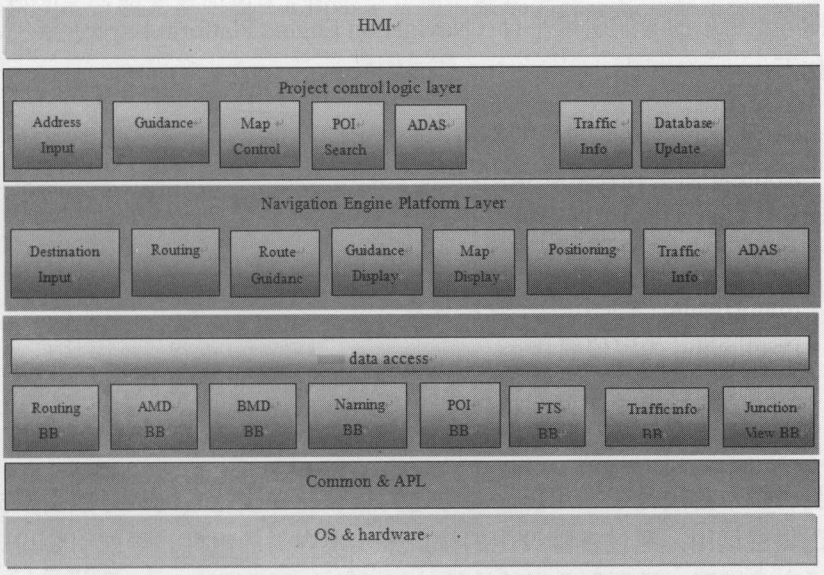


图 4-5　导航引擎的架构

如图 4-5 所示的项目层中，各模块的含义如下。

- HMI（Human Machine Interface）是人机界面的意思；
- Address Input 指位置输入模块；
- Guidance 为导航模块；
- Map Control 为地图显示模块；

- POI Search 为兴趣点（POI）搜索模块；
- ADAS 为自动辅助驾驶逻辑模块；
- Traffic Info 为交通信息模块，也可称为 telematics 模块；
- Database Update 为地图更新模块。

项目层包含 HMI 和业务逻辑层（project control logic layer），这些模块都是显示层面的模块，根据项目需求来定义。业务逻辑层在竖直方向上会通过导航引擎平台的接口调用实现导航的基本功能；在横向上，它在 speech（语音）、media（多媒体）、connectivity（蓝牙、Wi-Fi、Wlan、Telematics）等协同下进行应用整合。通过项目整合应用无缝连接车联网、多媒体、互联网 on-line 服务等，形成“本地导航、Off-board 导航、导航娱乐”三位一体的信息娱乐导航系统。从开发角度看，业务逻辑层因项目的需求（界面和流程）而变化，一般来说，界面对业务层的影响较大，但技术复杂度较小。

如图 4-5 所示的导航引擎平台（Navigation Engine Platform Layer）层中，各模块的含义如下。

- Destination Input 为目的地输入模块；
- Routing 为路径计算模块；
- Routing Guidance 为路径导航模块；
- Guidance Display 为导航显示逻辑模块；
- Map Display 为地图显示逻辑模块；
- Positioning 为定位模块；
- Traffic Info 为交通信息逻辑模块；
- ADAS 为自动辅助驾驶逻辑模块。

导航引擎平台层处于应用程序业务逻辑层和基础设施之间。导航引擎提供接口供业务逻辑层调用，通过基础设施的接口来实现内部的组件。导航引擎设计和定义自己的语言（模型、类型和数据结构等），以独立于任何第三方特定的平台和框架，同时借助于一套跨平台的框架实现一些基础的机制，是导航引擎平台化的基本思路。

如图 4-5 所示的数据获取层（data access ）中，各模块的含义如下。

- Routing BB 指道路数据的获取；
- AMD BB 是指 3D、DTM（数字高程图）、ORTHO（卫星图）等模块的数据获取；
- BMD BB 是指地图基本显示要素的获取；

- Naming BB 是指名称数据的获取；
- POI BB 是指兴趣点数据的获取；
- FTS BB 是指全文搜索数据库的获取；
- Traffic Info BB 是指交通信息数据的获取；
- Junction View BB 是指路口指示数据的获取。

这些获取数据模块作为数据层和逻辑层的接口，会提供一定的缓存机制，以减少引擎与程序的交互，提高引擎的反应速度。

如图 4-5 所示的基础设施（common & APL）层是一套跨平台的解决方案库和框架，能被引擎和业务逻辑层重用。比如：SQLite 库、IPC 异步通信和内存共享机制、GIS 库、OPENGLES 库等。

各模块间的依赖关系如图 4-6 所示。从通信架构上说，其关系如图 4-7 所示。

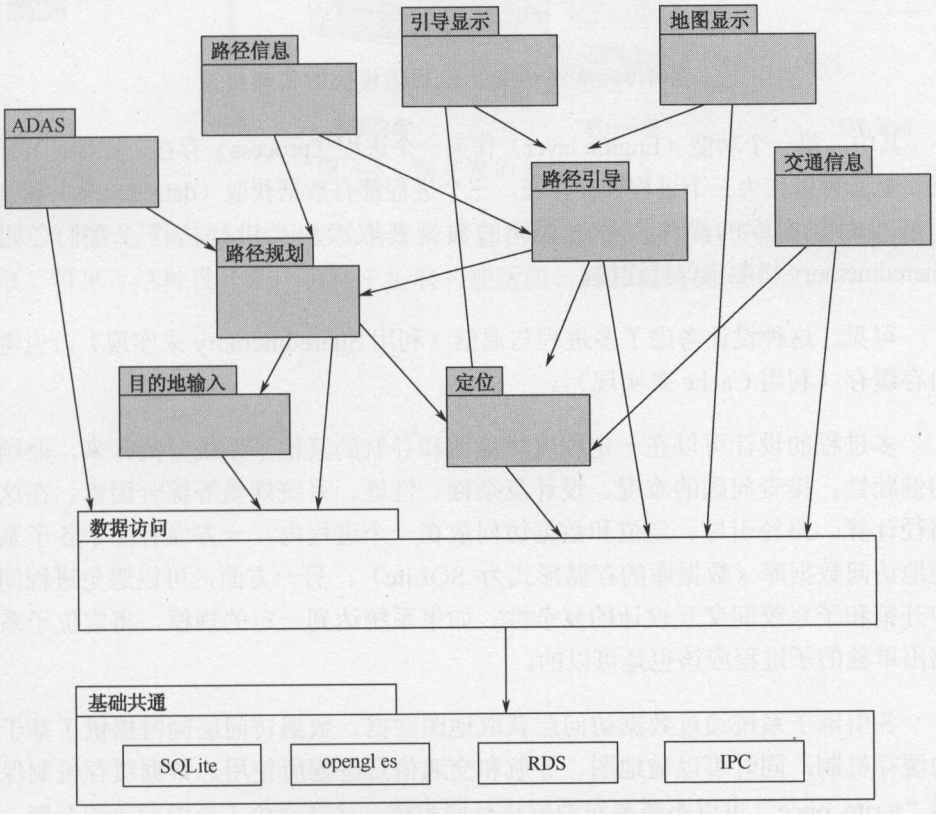


图 4-6　功能模块的依赖关系

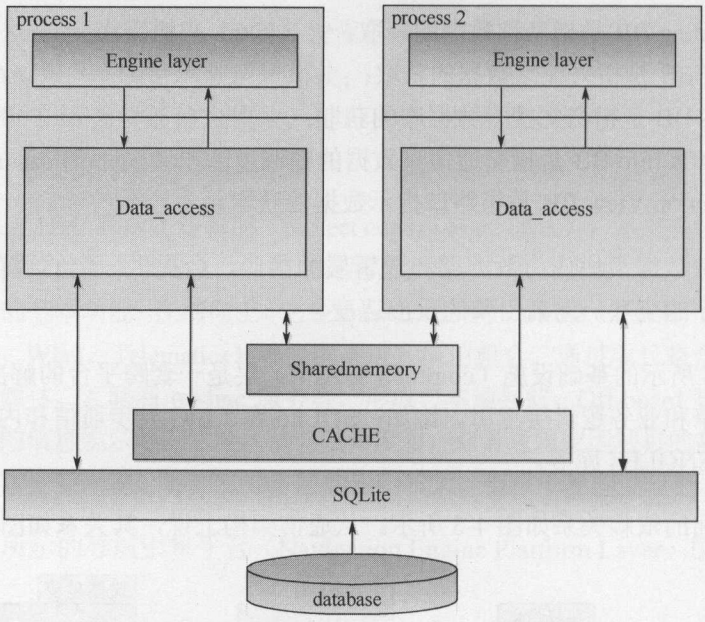


图 4-7　从通信架上看功能模块的依赖关系

其中，每一个功能（Engine layer）作为一个进程（process）存在，比如：导航、算路、显示可以作为三个进程独立存在，三个进程都有数据获取（data access）模块。为了减少对数据库的操作，各进程间的数据获取模块会进行协作，它们之间通过 Sharedmemory 机制来保持连接。

可见，这种设计考虑了多进程与通信（利用 Sharedmemory 来实现），也考虑了内存缓存（利用 Cache 来实现）。

多进程的设计可以在一定程度将地图和导航的其他子系统分离开来，兼顾系统的健壮性，排查问题的难度、设计复杂性、性能、系统规模等综合因素。在这里将路径计算、路径引导、定位和数据访问放在一个进程内，一方面有助于各子系统方便地访问数据库（数据库的存储形式为 SQLite），另一方面，可以避免进程间的同步开销和子系统间交互设计的复杂性。如果系统达到一定的规模，将定位子系统分离出单独的子进程应该也是可以的。

各引擎子系统通过数据访问层获取地图数据。数据访问层同时提供了基于内存的缓存机制，同时可以被地图、导航和交通信息进程所使用。数据缓存机制保证数据“write once”可以不需要对数据进行同步锁，这样避免了资源竞争的开销。对于路径计算，数据的访问将在引擎的上下文中执行，只有大数据没有命中时，才会去

SQLITE 中加载数据，然后分析数据流、适配数据对象，并完成写数据到共享缓存中。

如果是引擎部署在手机客户端的，获取地图数据需要利用网络通信，则可以在网络服务器端建立内存缓存，这样可以使手机客户端的用户通过手机端的引擎访问存储在网络服务器的内存缓存的数据，而不是访问网络服务器的数据库中的内容。

从硬件角度说，整体的架构如图 4-8 所示。（可参见 11.2 节的高并发部分）

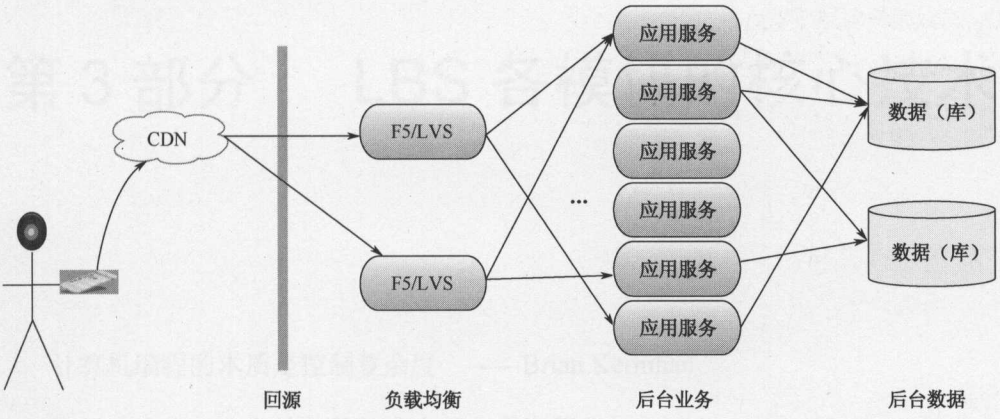


图 4-8　高并发的架构

由于这种设计能应付高并发的情况，而且在后期随着业务的发展方便进行维护和拓展。可见，这种设计是一种利于拓展的能适应、高并发的架构设计。

第 3 部分　LBS 各模块的核心技术

计算机编程的本质是控制复杂度　——Brian Kernihan

到目前为止，我们的数学技术大部分是微积分——Tom

对连续事物的分析有两种方法：几何方法（欧几里得、牛顿方法）和离散化方法（统计学、定积分、微积分、数论）。

第 5 章

数据处理

LBS 应用中常用的数据处理技术可以分为几何数据处理和图形处理。比如，LBS 应用中常见的文字数据等，虽然在处理上可以使用数据挖掘的知识，但是从一般性的数据处理技术上说，文字数据处理的核心技术在本质上属于几何数据中的点数据处理。

5.1 几何数据处理

几何是欧几里得、阿基米德、牛顿等数学大师最重视的数学技艺，比如：牛顿的《自然科学的数学原理》就是用几何学写成的。但在现代，几何学在一定程度上被忽视了，其重要性被低估了。所幸的是，LBS 领域在现代仍是几何学的一个重要阵地，比如：地图数据是最典型的一类几何数据，也是 LBS 的“虚拟现实”所在。在处理地图数据时，需要利用几何数据处理的技术。

三角剖分和化简也属于几何数据处理，但由于这两种技术与显示技术密切相关，所以不在本章中介绍，将在 8.1 节中介绍。

5.1.1 地图的结构

一份完整的地理数据（见图 5-1）往往包含如下模块。

图 5-1 地图外观

- BMD（基本显示模块）；
- 3D（三维建筑物）；
- POI（兴趣点）；
- ROUTE（道路）；
- JV（路口指示）；
- ADMIN（行政区划）；
- VOICE（语音）；
- ORTHO（卫星正交影像）；
- DTM（数字高程图）；
- TMC（实时交通信息）。

其中：

BMD 模块包含公园、水面、绿地、带高度的二维建筑物面等；

3D 模块包含三维建筑物；

POI 模块包含兴趣点、兴趣点的属性、兴趣点的深度信息；

ROUTE 模块包含道路及其形状点、道路的拓扑点、道路的属性信息和道路名称；

ADMIN 模块包含行政区划及其包含的区划面和名称信息；

JV 模块包含路口指示的图片、交叉路口的构成信息；

VOICE 模块包含语音指示的多媒体；

ORTHO 模块包含按照空间索引切分后的卫星图的图片；

DTM 模块包含按照空间索引切分后的数字高程图的图片；

TMC 模块包含实时交通信息及其与道路等的关联。

目前，市场上有很多地图数据格式，比如：NDS、KIWI、GDF、RDF、MIF 等。各种数据格式有的属于物理格式（即二进制字节流的数据，非明文），有的属于交换格式（即明文数据，可以用“记事本”直接打开），其组成并不相同，而且名称叫法各异，但是其根本的设计思想与模块的分类方法相同。

5.1.2 空间索引

空间索引是LBS数据处理的基础。不管是进行图像识别，还是激光点云或者地图数据处理，都需要构建数据索引。这是因为地图/图像的数据量太大，如果不建立空间索引，就会因为查询速度太慢而无法使用。由于实际的数据一般都会呈现簇状的聚类形态，所以，可以利用这种聚类来建立索引树，其基本思想是对搜索空间进行空间划分。目前常用的空间索引技术根据划分的空间是否有混叠，可以分为无重叠和有重叠两种。前者同层的索引结点（划分空间）没有重叠，其代表就是K-d树；后者同层的索引结点（划分空间）相互有交叠，其代表为R树。

1. K-d树索引

K-d树（K dimension tree）索引是一种对 k 维数据建立的树状索引，即这种方法可以对二维、三维或更多维的数据建立空间索引。

K-d树是一种平衡二叉树，也是一种空间划分树。对二维平面来说，具体的索引方法为：先对x（或者y） 进行二分，之后对y（或者x）进行二分，然后再对x（或者y）进行二分……

如果K-d树对二维平面建立空间索引，则这种K-d树索引又称为网格空间索引。

索引的建立方法如图5-2所示。

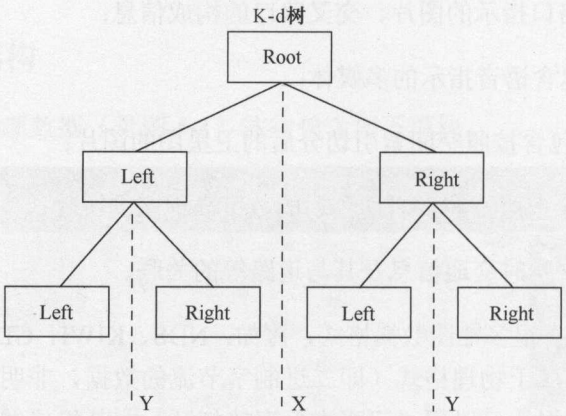


图5-2 K-d树索引

比如，有四个二维数据点{(2,3)，(9,6)，(4,7)，(7,2)}，数据点位于二维空间内，如图5-3所示。为了能有效地找到最近邻，K-d树采用分而治之的思想，即将整个空

间划分为几个小部分，首先，粗线将空间一分为二，然后在两个子空间中，细直线又将整个空间划分为四部分，最后虚线将其中一部分做进一步划分。

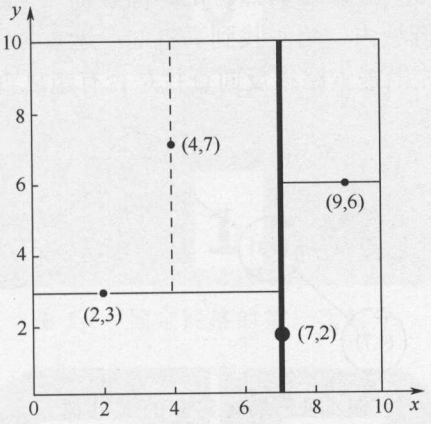


图 5-3　K-d 树的建立方法示例

四个二维数据点{(2,3)，(9,6)，(4,7)，(7,2)}构建 K-d 树的具体步骤如下。

1）利用 x 或者 y 进行划分（往往是利用方差大的坐标轴），比如：x。

2）确定 x 轴划分的中值。具体是：根据 x 维上的值将数据排序，四个数据的中值（所谓中值，即中间大小的值）为 4 或 7，所以这里选 7 的数据点（7,2）。这样，该结点的分割超平面就是通过（7,2），并垂直于 x 轴的直线 x=7。

3）确定：左子空间和右子空间。具体是：分割超平面 x=7 将整个空间分为两部分：$x \leqslant 7$ 的部分为左子空间，包含两个结点={(2,3)，(4,7)}；另一部分为右子空间，包含两个结点={(9,6) (7,2)}；

如上所述，K-d 树的构建是一个递归过程，我们对左子空间和右子空间内的数据重复根结点的过程，就可以得到一级子结点（2,3）和（9,6），同时将空间和数据集进一步细分，如此往复，直到空间中只包含一个数据点。

与此同时，经过对上面的空间进行划分之后，我们可以看出，点（7,2）可以为根结点，从根结点出发的两条斜线指向的（2,3）和（9,6）则为根结点的左右子结点，而（4,7）则为（2,3）的左孩子（也可设为右孩子），这样便形成了图 5-4 所示的 K-d 树。

对于 n 个实例的 k 维数据来说，建立 K-d 树的时间复杂度为 $O(knlogn)$。

在对 K-d 树进行最近点的查询时，比如：查询点（2.1,3.1），通过二叉搜索，顺着搜索路径很快就能找到最邻近的近似点，也就是叶子结点（2,3）。而找到的叶子结点并不一定就是最邻近的，最近的点肯定距离查询点更近，应该位于以查询点为圆心且通过叶子结点的圆域内。为了找到真正的最近点，还需要进行相关的回溯操作。也就是说，算法首先沿搜索路径反向查找是否有距离查询点更近的数据点。

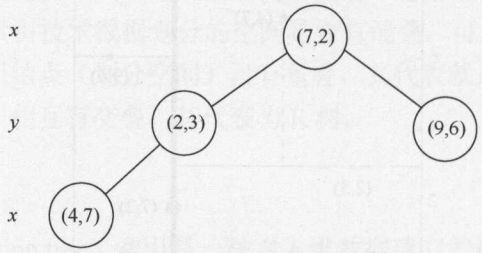


图 5-4　最终建立的 K-d 树

下面以查询（2.1,3.1）为例进行介绍。

1）二叉树搜索：先从（7,2）点利用 x 开始进行二叉查找，到达（2,3），此时搜索路径中的结点为<（7,2），（2,3）>，首先以（2,3）作为当前最近点，计算其到查询点（2.1,3.1）的距离为 0.14。

2）回溯查找：在得到（2,3）为查询点的最近点之后，回溯到其父结点（7,2），并判断在该父结点的其他子结点空间中是否有距离查询点更近的数据点。以（2.1,3.1）为圆心、以 0.14 为半径画圆，发现该圆并不与超平面 x= 7 交割，因此，不用进入（7,2）结点右子空间中去搜索。至此，搜索路径中的结点已经全部回溯完，结束整个搜索，返回最近的点（2,3），最近距离为 0.14。

以上网格空间索引的建立方法往往适用于对小数据的处理。对于大数据，考虑到数据的空间索引的建立效率，网格大小一般都是固定的，即：每一个索引结点都是同等大小的地理空间（通常称为：瓦片（Tile）或者网格（Mesh）），而不考虑结点内数据量的多少。

比如：如果先对地球按照 y 进行二分，并按照 0、1 编码，则全球地图会变成图 5-5 所示的形式。

之后再对地图按照 x 进行二分，并按照 0、1 编码，则地图在经过第二次划分后，生成了图 5-6 所示的第一层的四叉树。

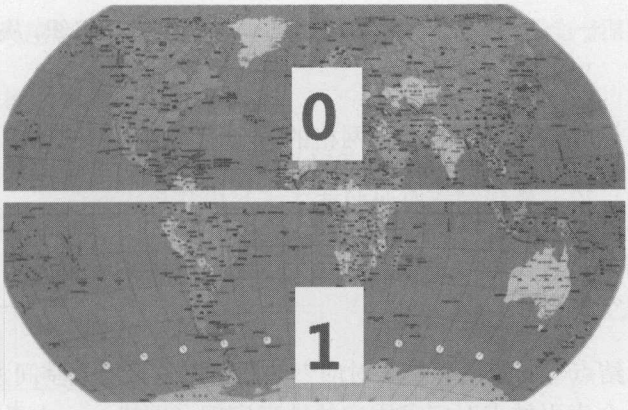


图 5-5　固定网格的第一次划分

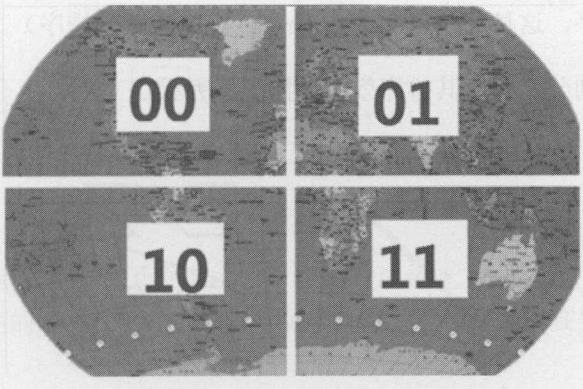


图 5-6　固定网格的第二次划分

利用这种目前最常用的按照大小恒定的地理空间来建立的空间索引的方法，进一步对地球进行划分，生成了第二层的四叉树索引，如图 5-7 所示。

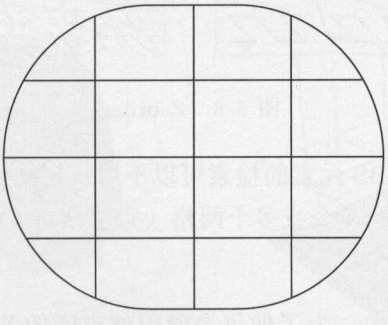


图 5-7　固定网格的第二层的四叉树

图 5-7 中的某一个瓦片都可以再按照四叉树进一步进行组织，从而建立整个空间索引。

可以看出，这幅地图已变为类似网格的一个四叉树索引的地图。所以，这种二维平面的索引又称为网格空间索引（将一幅地图数据按照网格划分，以落入每个网格内的地图目标建立索引）或四叉树索引（将已知范围的二维空间划成 4 个相等的子空间。如果需要，可以将每个或其中几个子空间继续划分下去，这样就形成了一个基于四叉树的空间划分）。

这种“索引结点与固定地理空间对应”的四叉树是目前的空间索引的主要形式。这种情况下，一个索引结点对应着固定的地理空间（网格）。由于在四叉树中每一层的结点都是有顺序的，所以，如果这种划分持续下去，则以上按照数字编码的各索引结点也有顺序，这种顺序就叫 Z-order（Z 排序或莫顿排序）。

按照不同的划分层级，其排序效果如图 5-8 所示。

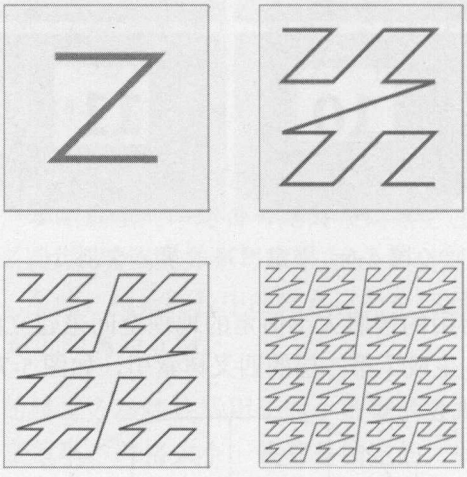


图 5-8　Z-order

在图 5-8 的顺序下，LBS 元素的检索可以不用从树的根结点开始寻找，而一般是根据地理空间所固定对应的一个或多个网格（索引结点），之后得到网格里的数据，再对数据进行排序或者挑选。

在目前工业界的地图中，为了使每个索引结点间的关系尽量简单，各个索引结点（网格）间的元素一般是被切割的，即：某个元素如果属于 a 网格，则不会再属于

b 网格。但是，跨越网格的地图元素不被切割也是可以应用 K-d 索引的。

下面以四叉树索引为例进行介绍。

一种跨越网格的元素不被切割的四叉树的结构如图 5-9 所示，地理空间对象都存储在叶子结点上，中间结点以及根结点不存储地理空间对象。可以看出，这种划分在空间上没有重叠，但是在元素上有重复（元素 2、4、8）。

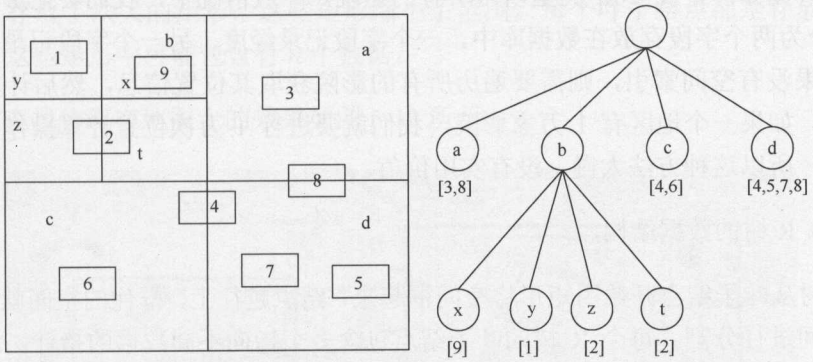


图 5-9　网格边缘不切割的四叉树

可见，这种四叉树方便了检索，由于在瓦片边缘无须切割，所以从本质上说，它与下面将要讲述的 R 树类似。对稀疏数据来说，这种四叉树在实际中也是有一定价值的。

K-d 树应用在多维空间的实例如下。

对一个三维空间来说，K-d 树按照一定的划分规则把这个三维空间划分为多个空间，如图 5-10、图 5-11 所示。

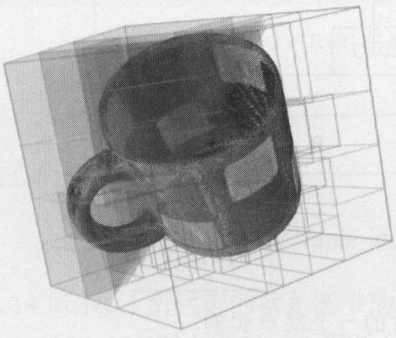


图 5-10　实体三维空间的 K-d 树

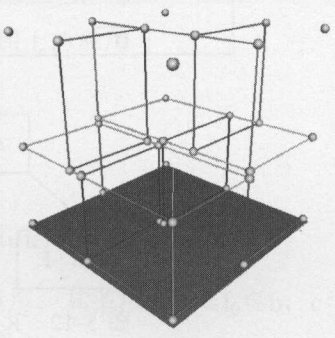


图 5-11　三维空间的划分示意图

2. R 树索引

R 树的构建思想是以最小边界矩形（Minimal Bounding Rectangle，简称 MBR）递归的对数据集空间进行划分。R 树中的非叶子结点代表划分的一个空间区域，即一个矩形空间区域；R 树中的叶子结点包含的矩形区域对应空间对象的 MBR。

R 树在数据库等领域做出的功绩是非常显著的，它很好地解决了在高维空间搜索等问题。比如：查找 200 英里以内所有的影院。一般情况下，我们会把影院的坐标（x,y）分为两个字段存放在数据库中，一个字段记录经度，另一个字段记录纬度。这样，如果没有空间索引，则需要遍历所有的影院获取其位置信息，然后计算是否满足要求。如果一个地区有 1 万家影院，我们就要进行 1 万次位置计算操作。由于数据量大，所以这种方法太慢，没有实用价值。

（1）R 树的数据结构

R 树从叶子结点开始用矩形将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。每个 R 树的叶子结点包含多个指向不同数据的指针，这些数据可以存放在硬盘中，也可以存放在内存中。根据 R 树的这种数据结构，当我们需要进行高维空间查询时，只需要遍历少数几个叶子结点所包含的指针，查看这些指针指向的数据是否满足要求即可。这种方式使我们不必遍历所有的数据就可以获得答案，效率显著提高。

图 5-12 是 R 树的一个简单实例。

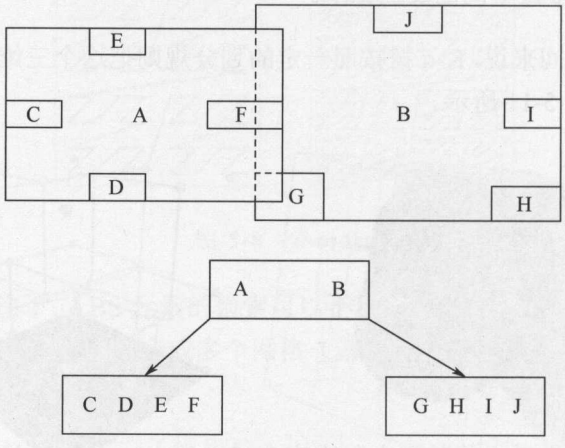


图 5-12　R 树的一个简单实例

用地图的例子来解释，由于要处理的数据都是影院或者某种类型的兴趣点，所以先把相邻的兴趣点划分到同一块区域。划分好所有的兴趣点之后，再把邻近的区域划分到更大的区域，划分完毕后再次进行更高层次的划分，直到只剩下两个最大的区域为止。

下面就可以把这些大小不同的矩形存入 R 树中。根结点存放的是两个最大的矩形，这两个最大的矩形框住了所有剩余的矩形，当然也就框住了所有的数据。下一层的结点存放了次大的矩形，这些矩形缩小了范围。每个叶子结点都是存放的最小的矩形，这些矩形中可能包含有 n 个数据。

R 树查找体现在地图上的效果如图 5-13 所示。

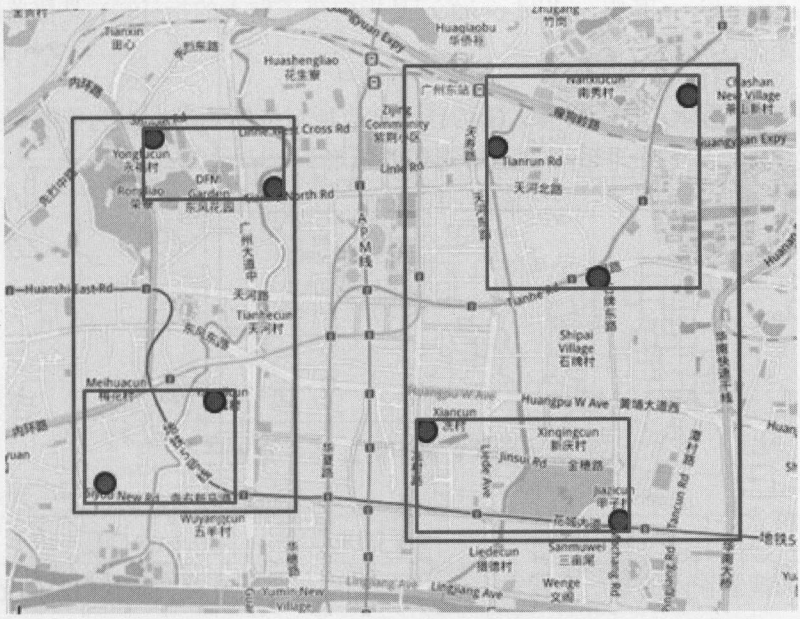


图 5-13　R 树在地图上的应用

（2）叶子结点的结构

叶子结点所保存的数据形式为：

（I，tuple-identifier）

如图 5-14 所示，I 所代表的就是图中的矩形，其范围是 $a \leqslant I_0 \leqslant b$，$c \leqslant I_1 \leqslant d$。有两个 tuple-identifier，在图中即表示为那两个黑点。

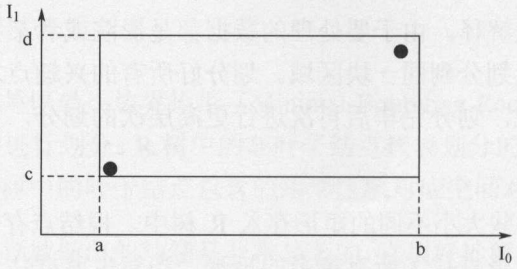


图 5-14　R 树叶子结点的结构

（3）非叶子结点

非叶子结点的结构与叶子结点非常类似，R 树的非叶子结点存放的数据结构为：（I，child-pointer）。其中，child-pointer 是指向孩子结点的指针，I 是覆盖所有孩子结点对应矩形的矩形。

如图 5-15 所示，D、E、F、G 为孩子结点所对应的矩形。A 为能够覆盖这些矩形的更大矩形，即这个非叶子结点所对应的矩形。

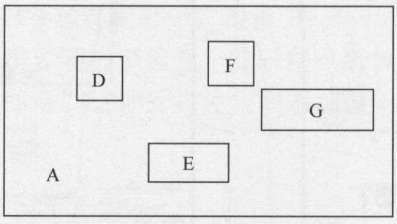


图 5-15　R 树中的非叶子结点

（4）搜索

R 树的搜索输入的是一个搜索矩形，返回的结果是所有符合查找信息的记录条目。

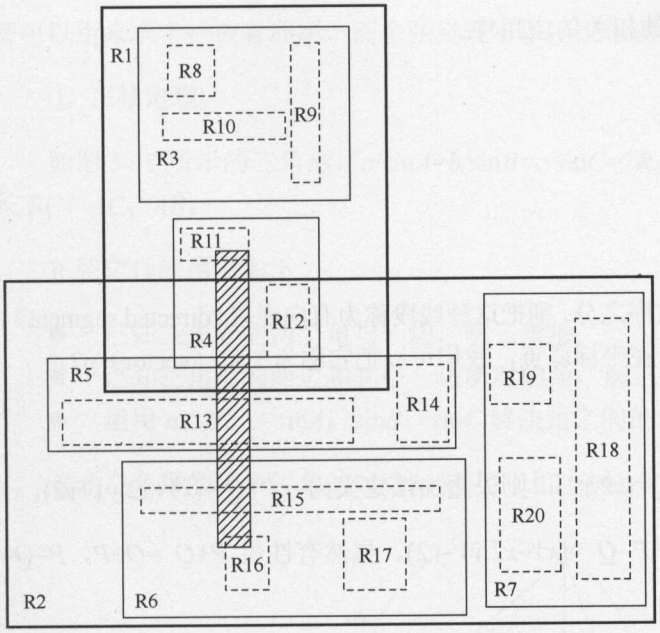
搜索的过程如下：

假设搜索的是图 5-16 所示的阴影部分所对应的矩形。

首先，需要搜索矩形在两个子树 R1、R2 上。

搜索 R1 后发现了 R1 中的 R4 矩形，继续搜索 R4，最终在 R4 所包含的 R11 与 R12 两个矩形中查找是否有符合条件的记录。

搜索 R2 的过程同样如此。很显然，该算法进行的是一个迭代操作。



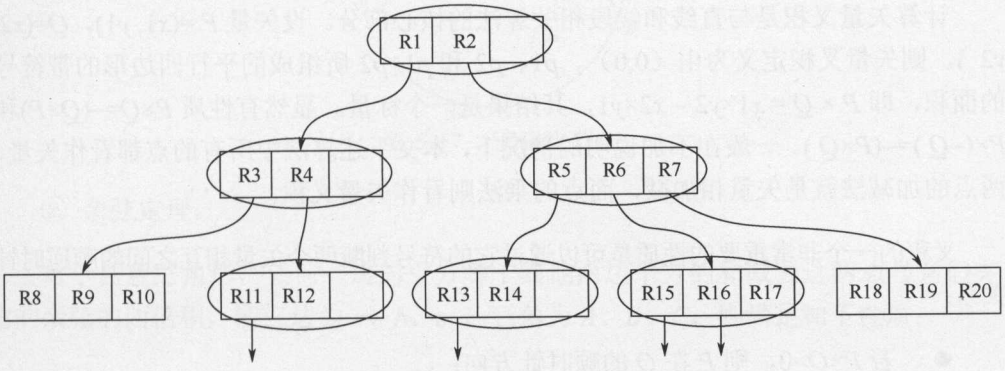


图 5-16　R 树搜索

在对 R 树进行添加、删除操作时，采用合并、分解结点的方法，以保证树的平衡性。

总的来说，R 树是叶子结点数量基本恒定的一种空间平衡树结构，也是能够有效地进行高维空间搜索的数据结构。它利用 B-树（平衡树）的思想来对多维空间进行分割。因此，可以认为，R 树就是一棵用来存储高维数据的平衡树。由于不需要像 K-d 树那样对网格元素切割就可以处理多维稠密数据，所以 R 树虽然类似于 K-d 树，但是由于在网片边缘元素可重叠，所以具备 K-d 树所不具备的优点，从而使它

被广泛应用在各种数据库及其相关的应用中。

5.1.3 几何图形

1. 矢量

（1）矢量的概念

如果一条线段的端点有次序之分，则把这种线段称为有向线段（directed segment）。如果有向线段 p1p2 的起点 p1 在坐标原点，我们可以把它称为矢量（vector）p2。

（2）矢量加减法

设二维矢量 $P=(x1,y1)$，$Q=(x2,y2)$，则矢量加法定义为：$P+Q=(x1+x2,y1+y2)$。

同理，矢量减法定义为：$P-Q=(x1-x2,y1-y2)$。显然有性质 $P+Q=Q+P$，$P-Q=-(Q-P)$。

（3）矢量叉积

计算矢量叉积是与直线和线段相关算法的核心部分。设矢量 $P=(x1, y1)$，$Q=(x2, y2)$，则矢量叉积定义为由（0,0）、$p1$、$p2$ 和 $p1+p2$ 所组成的平行四边形的带符号的面积，即 $P \times Q = x1 \times y2 - x2 \times y1$，其结果是一个标量。显然有性质 $P \times Q=-(Q \times P)$ 和 $P \times (-Q)=-(P \times Q)$。一般在不加说明的情况下，本文下述算法中所有的点都看作矢量，两点的加减法就是矢量相加减，而点的乘法则看作矢量叉积。

叉积的一个非常重要的性质是可以通过它的符号判断两个矢量相互之间的顺逆时针关系：

- 若 $P \times Q>0$，则 P 在 Q 的顺时针方向；
- 若 $P \times Q<0$，则 P 在 Q 的逆时针方向；
- 若 $P \times Q=0$，则 P 与 Q 共线，但可能同向，也可能反向。

2. 基本图形

基本图形有三种：三角形、圆和凸包。

（1）三角形

三角形是最基本的图形。三角形最重要的定理就是正余弦定理。比如，正弦定

理可以用来求半径或者曲率，而余弦定理可以用来求面积。

① 正弦定理。

如图 5-17 所示的三角形，$a/\sin A=b/\sin B=c/\sin C=2R$。其中，$a$、$b$、$c$ 分别对应边长 BC、AC、AB。

正弦定理的用途如下。

- 已知三角形的两个角与一边，解三角形；
- 已知三角形的两边和其中一边所对的角，解三角形；
- 运用 $a:b:c=\sin A:\sin B:\sin C$ 解决角之间的转换关系。

直角三角形的一个锐角的对边与斜边的比叫作这个角的正弦。

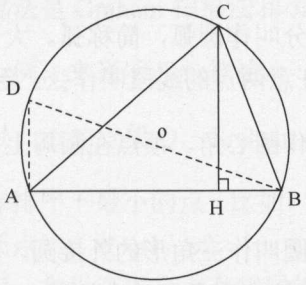


图 5-17　正弦定理

② 余弦定理。

对于任意三角形，任何一边的平方等于其他两边平方的和减去这两边与它们夹角的余弦的两倍积，若三边为 a、b、c，三角为 A、B、C，则满足如下性质：

$$a^2=b^2+c^2-2bc\cos A$$

$$b^2=a^2+c^2-2ac\cos B$$

$$c^2=a^2+b^2-2ab\cos C$$

$$\cos C=\frac{a^2+b^2-c^2}{2ab}$$

$$\cos B=\frac{a^2+c^2-b^2}{2ac}$$

$$\cos A=\frac{c^2+b^2-a^2}{2bc}$$

$$S_{\triangle ABC}=(1/2)ab\sin C$$

$$S_{\triangle ABC}=(1/2)bc\sin A$$

$$S_{\triangle ABC}=(1/2)ac\sin B$$

利用余弦定理可解决已知三角形两边及夹角求第三边，或者已知三条边求角的问题。

（2）圆

圆的基本性质如下。

① 平面上到定点的距离等于定长的所有点组成的图形叫作圆。定点称为圆心，定长称为半径。

② 圆上任意两点间的部分叫作圆弧，简称弧。大于半圆的弧称为优弧，小于半圆的弧称为劣弧。连接圆上任意两点的线段叫作弦。经过圆心的弦叫作直径。

③ 顶点在圆心上的角叫作圆心角。顶点在圆周上，且它的两边分别与圆有另一个交点的角叫作圆周角。

④ 过三角形三个顶点的圆叫作三角形的外接圆，其圆心叫作三角形的外心。与三角形的三边都相切的圆叫作这个三角形的内切圆，其圆心称为内心。

⑤ 直线与圆有三种位置关系：无公共点为相离；有两个公共点为相交；圆与直线有唯一公共点为相切，这条直线叫作圆的切线，这个唯一的公共点叫作切点。

⑥ 两个圆之间有五种位置关系：无公共点的，一个圆在另一个圆之外叫外离，在之内叫内含；有唯一公共点的，一个圆在另一个圆之外叫外切，在之内叫内切；有两个公共点的叫相交。两圆圆心之间的距离叫作圆心距。

⑦ 在圆上，由两条半径和一段弧围成的图形叫作扇形。圆锥侧面展开图是一个扇形，这个扇形的半径称为圆锥的母线。

（3）凸包

点集 Q 的凸包（convex hull）是指一个最小凸多边形，满足 Q 中的点或者在多边形边上，或者在其内。图 5-18 中的多边形就是其内部点集的凸包。

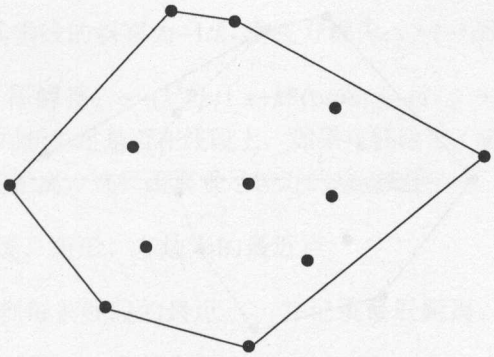


图 5-18　凸包

现在已经证明了凸包算法的时间复杂度下界是 $O(n\log n)$，但如果凸包的顶点数 h 也被考虑进去，Krikpatrick 和 Seidel 的剪枝搜索算法可以达到 $O(n\log n)$，在渐进意义下达到最优。最常用的凸包算法是 Graham 扫描法和 Jarvis 步进法。

下面以 Graham 扫描法为例，来说明凸包算法。

对于一个有三个或三个以上点的点集 Q，Graham 扫描法的伪代码过程如下：

先令 $p0$ 为 Q 中 Y–X 坐标排序下最小的点（比如，可以取最左点。如果有多个点是最左点，则应取其中的最低点）。设<$p1$，$p2$，…，pm>为对其余点按以 $p0$ 为中心的极角逆时针排序所得的点集，如果有多个点有相同的极角，除距 $p0$ 最远的点外全部移除；

```
压 p0 进栈 S;
压 p1 进栈 S;
压 p2 进栈 S;
for i ← 3 to m
do while 由 S 的栈顶元素的下一个元素、S 的栈顶元素以及 pi 构成的折线段不拐向左侧;
对 S 弹栈;
压 pi 进栈 S;
return S
```

上述过程执行后，栈 S 由底至顶的元素就是 Q 的凸包顶点按逆时针排列的点序列。寻找过程如图 5-19 所示。

需要注意的是，我们对点按极角逆时针排序时，并不需要真正求出极角，只需要求出任意两点的次序就可以。而这个步骤可以用前述的矢量叉积性质实现。

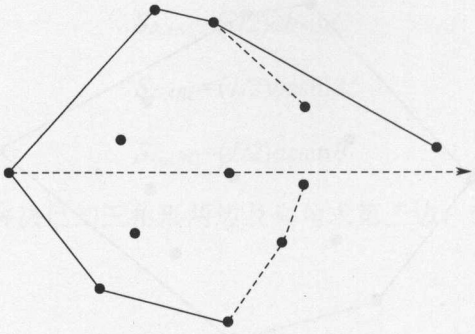


图 5-19　凸包建立过程

5.1.4　常用技巧

几何数据处理的常用技巧可以分为：点线关系、线线关系和点线面关系。

1. 点线关系

（1）判断点是否在线段上

设点为 Q，线段为 $P1P2$，判断点 Q 在该线段上的依据是：$(Q-P1)\times(P2-P1)=0$，且 Q 在以 $P1$、$P2$ 为对角顶点的矩形内。前者保证 Q 点在直线 $P1P2$ 上，后者是保证 Q 点不在线段 $P1P2$ 的延长线或反向延长线上，对这一步骤的判断可以用以下过程实现：

```
ON_LINK (pi,pj,pk)
if min(xi,xj) <= xk <= max(xi,xj) and min(yi,yj) <= yk <= max(yi,yj)
then return true;
else return false;
```

要特别注意的是，由于需要考虑水平线段和垂直线段两种特殊情况，因此，min(xi,xj)<=xk<=max(xi,xj)和 min(yi,yj)<=yk<=max(yi,yj)两个条件必须同时满足才能返回真值。

（2）计算点到线段的最近点

如果该线段平行于 X 轴（Y 轴），则过点 point 做该线段所在直线的垂线，垂足很容易求得，然后计算出垂足，如果垂足在线段上，则返回垂足，否则返回离垂足近的端点；如果该线段不平行于 X 轴，也不平行于 Y 轴，则斜率存在且不为 0。设线段的两端点为 $pt1$ 和 $pt2$，斜率为：$k=(pt2.y-pt1.y)/(pt2.x-pt1.x)$；该直线方程为：

$y=k*(x-pt1.x)+pt1.y$。其垂线的斜率为$-1/k$，垂线方程为：$y=(-1/k)*(x-\text{point}.x)+\text{point}.y$ 。

联合两个直线方程解得：$x=(k^2*pt1.x+k*(\text{point}.y-pt1.y)+\text{point}.x)/(k^2+1)$，$y=k*(x-pt1.x)+pt1.y$；然后判断垂足是否在线段上，如果在线段上，则返回垂足；如果不在，则计算两端点到垂足的距离，选择距离垂足较近的端点返回。

（3）计算点到折线、矩形、多边形的最近点

只要分别计算点到每条线段的最近点，并记录最近距离，然后取其中距离最小的点即可。

（4）计算点到圆的最近距离及交点坐标

如果该点在圆心，因为圆心到圆周任一点的距离相等，则返回 UNDEFINED。

连接点 P 和圆心 O，如果 PO 平行于 X 轴，则根据 P 在 O 的左边还是右边，计算出最近点的横坐标为 centerPoint.x – radius 或 centerPoint.x + radius。如果 PO 平行于 Y 轴，则根据 P 在 O 的上边还是下边，计算出最近点的纵坐标为 centerPoint.y + radius 或 centerPoint.y – radius。如果 PO 不平行于 X 轴和 Y 轴，则 PO 的斜率存在且不为 0，这时直线 PO 斜率为 $k = (P.y-O.y)/(P.x-O.x)$。直线 PO 的方程为：$y = k * (x-P.x)+P.y$。设圆方程为：$(x-O.x)^2 + (y-O.y)^2 = r^2$，联立两个方程组可以解出直线 PO 和圆的交点，取其中离 P 点较近的交点即可。

2. 线线关系

（1）折线段的拐向判断

折线段的拐向判断方法可以直接由矢量叉积的性质推出。对于有公共端点的线段 $p0p1$ 和 $p1p2$，通过计算$(p2-p0)\times(p1-p0)$的符号便可以确定折线段的拐向：

若$(p2-p0)\times(p1-p0)>0$，则 $p0p1$ 在 $p1$ 点拐向右侧后得到 $p1p2$。

若$(p2-p0)\times(p1-p0)<0$，则 $p0p1$ 在 $p1$ 点拐向左侧后得到 $p1p2$。

若$(p2-p0)\times(p1-p0)=0$，则 $p0$、$p1$、$p2$ 三点共线。

（2）判断两条线段是否相交

判断两条线段是否相交可以分为以下两步。

1）快速排斥试验。

假设以线段 $P1P2$ 为对角线的矩形为 R，以线段 $Q1Q2$ 为对角线的矩形为 T，如果 R 和 T 不相交，则两线段不会相交。

2）跨立试验。

如果两条线段相交，则两条线段必然相互跨立对方。若 $P1P2$ 跨立 $Q1Q2$，则矢量（$P1-Q1$）和（$P2-Q1$）位于矢量（$Q2-Q1$）的两侧，即$((P1-Q1)\times(Q2-Q1)) * ((P2-Q1)\times(Q2-Q1))< 0$。上式可改写成$((P1-Q1)\times(Q2-Q1))*((Q2-Q1)\times(P2-Q1))> 0$。当$(P1-Q1)\times(Q2-Q1) = 0$ 时，说明（$P1-Q1$)和($Q2-Q1$)共线，但是因为已经通过快速排斥试验，所以，$P1$ 一定在线段 $Q1Q2$ 上；同理，$(Q2-Q1)\times(P2-Q1) = 0$，说明 $P2$ 一定在线段 $Q1Q2$ 上。所以判断 $P1P2$ 跨立 $Q1Q2$ 的依据是：$((P1-Q1)\times(Q2-Q1)) * ((Q2-Q1)\times(P2-Q1))\geqslant 0$。同理，判断 Q1Q2 跨立 P1P2 的依据是：((Q1–P1) ×(P2 –P1)) * ((P2 – P1) ×(Q2 –P1))≥ 0。

（3）判断线段和直线是否相交

有了上面的基础，这个算法就很容易了。如果线段 $P1P2$ 和直线 $Q1Q2$ 相交，则 $P1P2$ 跨立 $Q1Q2$，即：$((P1-Q1)\times(Q2-Q1)) *((Q2-Q1) \times(P2 -Q1)) \geqslant 0$。

（4）计算两条共线的线段的交点

对于两条共线的线段，它们之间的位置关系有以下几种情况：

- 两条线段没有交点；
- 两条线段有无穷个交点；
- 两条线段有一个交点。

设 line1 是两条线段中较长的一条，line2 是较短的一条，如果 line1 包含 line2 的两个端点，则两条线段有无穷个交点；如果 line1 只包含 line2 的一个端点，那么如果 line1 的某个端点等于被 line1 包含的 line2 的那个端点，则两条线段只有一个交点，否则就是两条线段也有无穷个交点；如果 line1 不包含 line2 的任何端点，则两条线段没有交点。

（5）计算线段或直线与线段的交点

设一条线段为 $L0 = P1P2$，另一条线段或直线为 $L1=Q1Q2$，要计算的就是 $L0$ 和

*L*1 的交点。

1）判断 *L*0 和 *L*1 是否相交，如果不相交则没有交点，否则说明 *L*0 和 *L*1 一定有交点，下面就将 *L*0 和 *L*1 都当作直线来考虑。

2）如果 *P*1 和 *P*2 的横坐标相同，即 *L*0 平行于 *Y* 轴。

① 若 *L*1 也平行于 *Y* 轴，且 *P*1 的纵坐标和 *Q*1 的纵坐标相同，说明 *L*0 和 *L*1 共线，假如 *L*1 是直线，则它们有无穷个交点，假如 *L*1 是线段，则可用“计算两条共线线段的交点”的算法求它们的交点；否则说明 L0 和 L1 平行，它们没有交点。

② 若 *L*1 不平行于 *Y* 轴，则交点横坐标为 *P*1 的横坐标，代入 *L*1 的直线方程中可以计算出交点纵坐标。

3）如果 *P*1 和 *P*2 横坐标不同，但是 *Q*1 和 *Q*2 横坐标相同，即 *L*1 平行于 *Y* 轴，则交点横坐标为 *Q*1 的横坐标，代入 *L*0 的直线方程中可以计算出交点纵坐标。

4）如果 *P*1 和 *P*2 纵坐标相同，即 *L*0 平行于 *X* 轴。

① 若 *L*1 也平行于 *X* 轴，且 *P*1 的横坐标和 *Q*1 的横坐标相同，说明 *L*0 和 *L*1 共线，假如 *L*1 是直线，则它们有无穷个交点，假如 *L*1 是线段，则可用“计算两条共线线段的交点”的算法求它们的交点；否则说明 *L*0 和 *L*1 平行，它们没有交点。

② 若 *L*1 不平行于 *X* 轴，则交点的纵坐标为 *P*1 的纵坐标，代入 *L*1 的直线方程中可以计算出交点的横坐标。

5）如果 *P*1 和 *P*2 纵坐标不同，但是 *Q*1 和 *Q*2 纵坐标相同，即 *L*1 平行于 *X* 轴，则交点的纵坐标为 *Q*1 的纵坐标，代入 *L*0 的直线方程中可以计算出交点的横坐标。

6）剩下的情况就是 *L*1 和 *L*0 的斜率均存在且不为 0 的情况。

① 计算出 *L*0 的斜率 *K*0，*L*1 的斜率 *K*1。

② 如果 *K*1=*K*2，且 *Q*1 在 *L*0 上，则说明 *L*0 和 *L*1 共线，假如 *L*1 是直线，则有无穷个交点，假如 *L*1 是线段，则可用“计算两条共线线段的交点”的算法求它们的交点；如果 *Q*1 不在 *L*0 上，则说明 *L*0 和 *L*1 平行，它们没有交点。

③ 联立两个直线方程组可以解出交点。

这个算法并不复杂，但是要分情况讨论清楚，尤其是当两条线段共线的情况需

要单独考虑，所以在前文将求两条共线线段的算法单独写出来。另外，一开始就应利用矢量叉乘判断线段与线段（或直线）是否相交，如果结果相交，那么在后面就可以将线段全部看作直线来考虑。需要注意的是，我们可以将直线或线段方程改写为 $ax+by+c=0$ 的形式，这样，上述过程的部分步骤可以合并，缩短代码长度，但是由于先要求出参数，这种算法将花费更多的时间。

（6）求线段或直线与折线、矩形、多边形的交点

分别求与每条边的交点即可。

（7）求线段或直线与圆的交点

设圆心为 O，圆半径为 r，直线（或线段）L 上的两点为 $P1$ 和 $P2$。

1）如果 L 是线段且 $P1$、$P2$ 都包含在圆 O 内，则没有交点；否则进行下一步。

2）如果 L 平行于 Y 轴，则有

① 计算圆心到 L 的距离 dis。

② 如果 $dis>r$，则 L 和圆没有交点。

③ 利用勾股定理可以求出两个交点坐标。但要注意考虑 L 和圆的相切情况。

3）如果 L 平行于 X 轴，做法与 L 平行于 Y 轴的情况类似。

4）如果 L 既不平行 X 轴，也不平行 Y 轴，可以求出 L 的斜率 K，然后列出 L 的点斜式方程，与圆方程联立，即可求解出 L 和圆的两个交点。

5）如果 L 是线段，对步骤 2）～步骤 4）中求出的交点，还要分别判断是否属于该线段的范围内。

3. 点线面的关系

（1）判断矩形是否包含点

只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间即可。

（2）判断点是否在圆内

计算圆心到该点的距离，如果小于或等于半径，则该点在圆内。

（3）判断点是否在多边形中

判断点 P 是否在多边形中是计算几何中一个非常基本但又十分重要的算法。以点 P 为端点，向左方绘制射线 L，由于多边形是有界的，所以射线 L 的左端一定在多边形外，考虑沿着 L 从无穷远处开始自左向右移动，遇到和多边形的第一个交点时，进入多边形的内部，遇到第二个交点时，离开多边形，所以很容易看出，当 L 和多边形的交点数目 C 是奇数时，P 在多边形内，否则 P 在多边形外。

但有以下特殊情况要加以考虑。

L 和多边形的顶点相交，这时交点只能计算一个；
L 和多边形的一条边重合，这条边应该被忽略不计。

在计算射线 L 和多边形的交点时，做如下限定：

对多边形的水平边不作考虑；

对多边形的顶点和 L 相交的情况，如果该顶点是其所属的边上纵坐标较大的顶点，则计数，否则忽略；

对 P 在多边形边上的情形，可直接判断 P 属于多边行。

由此得出算法的伪代码如下：

```
count ← 0;
以 P 为端点，作从右向左的射线 L;
for 多边形的每条边 s
do if P在边 s 上
then return true;
if s 不是水平的
then if s 的一个端点在 L 上
if 该端点是 s 两端点中纵坐标较大的端点
then count ← count+1
else if s 和 L 相交
then count ← count+1;
if count mod 2 = 1
then return true;
else return false;
```

其中，做射线 L 的方法是：设 P'的纵坐标和 P 相同，横坐标为正无穷大（很大的一个正数），则 P 和 P'就确定了射线 L。

判断点是否在多边形中的这个算法的时间复杂度为 $O(n)$。

另外，还有一种算法是用带符号的三角形面积（点与多边形的每条边组成）之和与多边形面积进行比较，这种算法由于使用浮点数运算，所以会带来一定的误差。

（4）判断线段是否在多边形内

线段在多边形内的一个必要条件是线段的两个端点都在多边形内，但由于多边形可能为凹，所以这不能成为判断的充分条件。如果线段和多边形的某条边内交（两线段内交是指两线段相交且交点不在两线段的端点），因为多边形的边的左右两侧分属多边形内外不同的部分，所以线段会有一部分在多边形外。在这里得到了线段在多边形内的第二个必要条件：线段和多边形的所有边都不内交。

线段和多边形交于线段的两端点并不会影响线段是否在多边形内，但如果多边形的某个顶点和线段相交，还必须判断两相邻交点之间的线段是否包含于多边形内部。

因此，可以先求出所有与线段相交的多边形的顶点，然后按照 X–Y 坐标排序（X 坐标小的排在前面，对于 X 坐标相同的点，Y 坐标小的排在前面，这种排序准则也是为了保证水平和垂直情况的判断正确），这样相邻的两个点就是在线段上相邻的两个交点，如果任意相邻两点的中点也在多边形内，则该线段一定在多边形内。

设多边形和线段 PQ 的交点依次为 P_1，P_2，…，P_n，其中 P_i 和 P_{i+1} 是相邻的两个交点，线段 PQ 在多边形内的充要条件是：P、Q 在多边形内，且对于 i=1、2、…、n–1，P_i 与 P_{i+1} 的中点也在多边形内。

在实际编程中，没有必要计算所有的交点，首先应判断线段和多边形的边是否内交，若线段和多边形的某条边内交，则线段一定在多边形外；如果线段和多边形的每一条边都不内交，则线段和多边形的交点一定是线段的端点或者多边形的顶点，只要判断点是否在线段上就可以。

至此得出算法如下：

```
if 线端 PQ 的端点不都在多边形内
then return false;
点集 pointSet 初始化为空;
for 多边形的每条边 s
do if 线段的某个端点在 s 上
then 将该端点加入 pointSet;
```

```
else if s 的某个端点在线段 PQ 上
then 将该端点加入 pointSet;
else if s 和线段 PQ 相交 // 这时候可以肯定是内交了
then return false;
将 pointSet 中的点按照 X-Y 坐标排序;
for pointSet 中每两个相邻点 pointSet[i] , pointSet[i+1]
do if pointSet[i] , pointSet[i+1] 的中点不在多边形中
then return false;
return true;
```

这个过程中的排序因为交点数目肯定远小于多边形的顶点数目 n，所以最多是常数级的复杂度，几乎可以忽略不计。因此，算法的时间复杂度也是 $O(n)$。

（5）判断折线是否在多边形内

只要判断折线的每条线段是否都在多边形内即可。设折线有 m 条线段，多边形有 n 个顶点，则该算法的时间复杂度为 $O(mn)$。

（6）判断线段、折线、多边形是否在矩形中

因为矩形是一个凸集，所以只要判断所有的端点是否都在矩形中就可以。

（7）判断矩形是否在矩形中

只要比较左右边界和上下边界就可以。

（8）判断矩形是否在多边形内

将矩形转化为多边形，然后判断是否在多边形内。

（9）判断线段、折线、矩形、多边形是否在圆内

因为圆是凸集，所以只要判断是否每个顶点都在圆内即可。

（10）判断圆是否在圆内

设两个圆为 O1、O2，半径分别为 $r1$、$r2$，要判断 O2 是否在 O1 内，首先要比较 $r1$、$r2$ 的大小，如果 $r1<r2$，则 O2 不可能在 O1 内；否则，如果两圆心的距离大于 $r1-r2$，则 O2 不在 O1 内；否则 O2 在 O1 内。

（11）判断圆是否在矩形中

很容易证明，圆在矩形中的充要条件是：圆心在矩形中且圆的半径小于或等于

圆心到矩形四边的距离的最小值。

（12）判断圆是否在多边形内

只要计算圆心到多边形的每条边的最短距离即可，如果该距离大于或等于圆半径，则该圆在多边形内。计算圆心到多边形每条边最短距离的算法在后文阐述。

（13）判断多边形是否在多边形内

只要判断多边形的每条边是否都在多边形内即可。判断有 m 个顶点的一个多边形是否在有 n 个顶点的一个多边形内的复杂度为 $O(mn)$。

5.2 图像处理

图像处理是 LBS 的“彩头”。一个 LBS 应用能否出彩，很大程度上取决于图像处理技术的合理运用。比如：

- 微信的语音、图片和视频的体积都很小，很省手机流量，这是微信深受人们喜欢的一个重要原因；
- 很多图像分享应用允许用户做很多滤镜处理，能使用户的图像看起来更漂亮，深受粉丝的喜爱。

以上示例的具体技术分析如下。

（1）微信

语音压缩方面，可以利用微信通过声音的频率来压缩后，再在网络中传输压缩后的数据，之后在接收端的微信来解压缩。

图片压缩方面，可以利用傅里叶频域变换后，利用各种低通滤波器进行滤波。在进行网络传输时，只传输频域的信息就可以。

视频压缩方面，视频可以认为是多帧的图像，之后对每一帧进行压缩处理，也可以利用视频中不同帧的不同像素进行对比，从而在第一帧存储原始图像，后面的帧只存储相对于第一帧的差异像素。

（2）使用图像滤镜的应用

往往是利用线性滤波器，利用某个掩模算子对图像进行处理，或者利用某个颜

色范围对图像的像素进行修改，从而使图像具有复古、青春等色彩。

总的来说，图像处理也属于数据处理，所以，在数据处理、数据挖掘，或网络传输的压缩技术章节的知识，也可以用于图形处理。除这些通用技术外，图形处理还有一些专门的技术，比如：傅里叶变换、线性滤波器。

5.2.1 傅里叶变换

1. 信号（或图像、数据）在频域中的意义

在频域中，频率越大，说明原始信号变化的速度越快；频率越小，说明原始信号越平缓。当频率为 0 时，表示直流信号，没有变化。因此，频率的大小反映信号变化的快慢。高频分量解释信号的突变部分，而低频分量决定信号的整体形象。

在图像处理中，频域反映图像在空域中灰度变化的剧烈程度，也就是图像灰度的变化速度或者是图像的梯度大小。对图像而言，图像的边缘部分是突变部分，变化较快，因此，反映在频域上是高频分量；图像的噪声在大部分情况下是高频部分；图像平缓变化部分则为低频分量。也就是说，傅里叶变换提供另外一个角度来观察图像，可以将图像从灰度分布转化到频率分布上观察图像的特征。

傅里叶变换是将时域信号分解为不同频率的正弦信号或余弦函数叠加之和。在连续情况下，要求原始信号在一个周期内满足绝对可积条件，傅里叶变换才会成立。而在离散情况下，傅里叶变换一定存在。

2. 傅里叶变换的一个比喻

一个恰当的比喻是将傅里叶变换比作一个玻璃棱镜。棱镜是可以将光分解为不同颜色的物理仪器，每个成分的颜色由波长（或频率）来决定。傅里叶变换可以看作是数学上的棱镜，将函数基于频率分解为不同的成分。当我们考虑光时，讨论它的光谱或频率谱。同样，傅里叶变换使我们能通过频率成分来分析一个函数。

3. 傅里叶变换的定义

① 傅里叶变换

$$F(\omega)=\int_{-\infty}^{\infty} f(t)\mathrm{e}^{-iwt}\mathrm{d}t \qquad \text{（式 5-1）}$$

② 傅里叶逆变换

$$f(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}F(\omega)\mathrm{e}^{-iwt}\mathrm{d}\omega \qquad \text{（式 5-2）}$$

$f(t)$是 t 的周期函数，$f(t)$的狄里赫莱条件如下：

- 在一个以 $2T$ 为周期内 $f(t)$连续或只有有限个第一类间断点；
- $f(t)$ 单调或可划分成有限个单调区间；
- 在一个周期内具有有限个极值点；
- 绝对可积。

如果 $f(t)$满足狄里赫莱条件，则有（式 5-1）成立，称为积分运算 $f(t)$的傅里叶变换。

（式 5-2）的积分运算叫作 $F(\omega)$的傅里叶逆变换。

4. 傅里叶变换的升级：小波变换

首先，使用正弦波和余弦波，理论上可以叠加为一个矩形。

任何波形都可以按照这种方法用正弦波叠加起来，如图 5-20 所示。

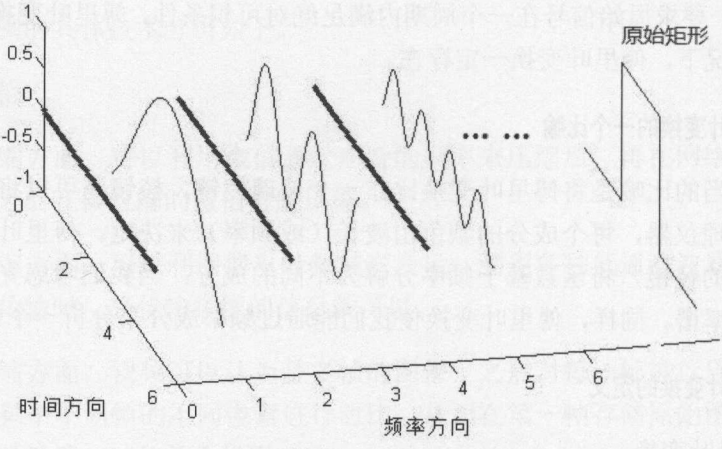


图 5-20　正弦波叠加得到任意波

通过图 5-20 可以发现，在频谱中，偶数项的振幅都是 0，也就对应了图 5-20 中振幅为 0 的正弦波。

5. 傅里叶变换的应用

傅里叶变换在图像处理中有非常重要的作用。因为不仅傅里叶分析涉及图像处理的很多方面，傅里叶的改进算法，比如离散余弦变换、gabor 与小波在图像处理中也很重要。总的来说，傅里叶变换会应用在图像处理的以下领域。

（1）图像增强与图像去噪

绝大部分噪声都是图像的高频分量，通过低通滤波器来滤除高频（噪声）；边缘也是图像的高频分量，可以通过添加高频分量来增强原始图像的边缘。

（2）图像特征提取

纹理特征：直接通过傅里叶系数来计算纹理特征。

（3）图像压缩

可以直接通过傅里叶系数来压缩数据；常用的离散余弦变换是傅里叶变换的实变换。

关于图像滤波的几个概念如下。

- 图像高频分量：图像突变部分，在某些情况下指图像边缘信息，某些情况下指噪声，更多的是两者的混合；
- 低频分量：图像变化平缓的部分，也就是图像轮廓信息；
- 高通滤波器：让图像使低频分量抑制，高频分量通过；
- 低通滤波器：与高通相反，让图像使高频分量抑制，低频分量通过；
- 带通滤波器：使图像在某一部分的频率信息通过，其他过低或过高的频率都抑制；
- 带阻滤波器与带通滤波器相反。

5.2.2 线性滤波器

拉普拉斯算子的知识是线性滤波器的基础。拉普拉斯算子是最简单的各向同性微分算子，具有旋转不变性。一个二维图像函数的拉普拉斯算子是各向同性的二阶导数，定义为：

$$\nabla^2 f(x,y)=\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}$$

为了更适合做数字图像处理，将图像看作是一个离散值的函数后，可将该算子表示为离散形式：

$$\nabla^2 f=\left[f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)\right]-4f(x,y)$$

由于图像中的每一个像素都可以代表上面公式中的一个 $f()$值，所以在进行图像处理时，拉普拉斯算子还可以表示成模板的形式，如图 5-21 所示。

0	1	0
1	−4	1
0	1	0

（a）拉普拉斯运算模板

1	1	1
1	−8	1
1	1	1

（b）拉普拉斯运算扩展模板

图 5-21　拉普拉斯算子

从模板形式容易看出，如果在图像的一个较暗区域中出现了一个亮点，那么用拉普拉斯运算就会使这个亮点变得更亮。因为图像中的边缘就是那些灰度发生跳变的区域，所以，拉普拉斯锐化模板在边缘检测中很有用。一般增强技术对于陡峭的边缘和缓慢变化的边缘很难确定其边缘线的位置。但此算子却可用二次微分正峰和负峰之间的过零点来确定，对孤立点或端点更敏感，因此，特别适用于以突出图像中的孤立点、孤立线或线端点为目的的场合。这在本质上是对图像的一种锐化。

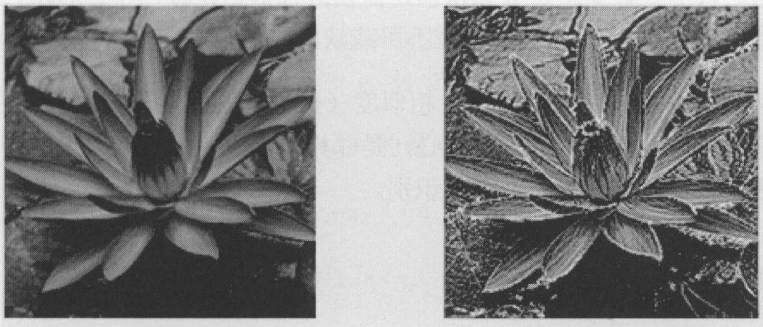
图像锐化处理的作用是使灰度反差增强，从而使模糊图像变得更加清晰。图像模糊的实质就是对图像进行平均运算或积分运算，因此，可以对图像进行逆运算，如微分运算能够突出图像细节，使图像变得更清晰。由于拉普拉斯是一种微分算子，它的应用可增强图像中灰度突变的区域，减弱灰度的缓慢变化区域。因此，锐化处理可选择拉普拉斯算子对原图像进行处理，产生描述灰度突变的图像，再将拉普拉斯图像与原始图像叠加产生锐化图像。拉普拉斯锐化的基本方法可以由下式表示：

$$g(x,y)=f(x,y)-\nabla^2 f(x,y)\quad 或\quad g(x,y)=f(x,y)+\nabla^2 f(x,y)$$

这种简单的锐化方法既可以产生拉普拉斯锐化处理的效果，同时又能保留背景信息，将原始图像叠加到拉普拉斯变换的处理结果中，可以使图像中的各灰度值得到保留，使灰度突变处的对比度得到增强，最终结果是在保留图像背景的前提下，

突现出图像中小的细节信息。

如图 5-22 所示是拉普拉斯模板对图像滤波后的实验效果。可以看出，将原始图像通过拉普拉斯变换后增强了图像中灰度突变处的对比度，使图像中小的细节部分得到增强并保留图像的背景色调，使图像的细节比原始图像更加清晰。基于拉普拉斯变换的图像增强已成为图像锐化处理的基本工具。



（a）原始图像　　（b）锐化后的图像

图 5-22　锐化

上面是以锐化来说明狭义的拉普拉斯算子工作原理。实际上，泛义的拉普拉斯算子可以用来做锐化，也可以用来做模糊化，或者其他滤波，只是模板中的具体数值与狭义拉普拉斯算子有所不同罢了。

由于拉普拉斯算子在概念上的重要性，我们通常称泛义的拉普拉斯算子运算模板为：滤波器、掩模、滤波掩模、核、模板或窗口。

这些名字和叫法各有不同，但其本质上与狭义的拉普拉斯算子是一样的。

第 6 章

数据挖掘

大数据时代的大数据本身并无价值，需要进行数据挖掘，才会对用户产生价值，所以数据挖掘已经成为目前 LBS 领域不可或缺的重要技术之一。

数据挖掘可以从三个维度来说：相似度（数据挖掘的基础技术，往往可用于推荐技术）、分类技术（数据挖掘的核心，是以相似度技术为基础来进行分类的技术，可用于预测、识别、推荐等）和图像识别。

6.1 相似度

相似度有两个重要的度量：距离和相关系数。其中，距离用来表示两组散乱数据间的相似度；而相关系数用来表示两组近似线性的数据的相似度。由于相似度技术是推荐系统的主要技术之一，所以，关于相似度的技术知识也是 9.2 节的基础。

6.1.1 距离

距离是分类的基础，也是数据挖掘中最重要的概念之一，它是衡量相似度的主要指标之一。主要的距离概念可以分为两类：一般意义上的物理距离（闵可夫斯基距离，这种距离一般是从几何上直观可见的）、与概率统计相关的距离。

1. 闵可夫斯基距离（Minkowski Distance）

两个 n 维变量 $a(x_{11},x_{12},\cdots,x_{1n})$与 $b(x_{21},x_{22},\cdots,x_{2n})$间的闵可夫斯基距离定义为：

$$d_{12}=\sqrt[p]{\sum_{k=1}^{n}|x_{1k}-x_{2k}|^{p}}$$

其中，p 是一个变参数。

当 p=1 时，就是曼哈顿距离；

当 p=2 时，就是欧氏距离；

当 $p\to\infty$ 时，就是切比雪夫距离。

根据变参数的不同，闵氏距离可以表示一类距离。

（1）曼哈顿距离

曼哈顿距离的正式意义为闵可夫斯基距离的 $p=1$ 时的距离，或城市区块距离（City Block distance），也就是在欧几里得空间的固定直角坐标系上两点所形成的线段对轴产生的投影的距离总和。例如，在平面上，坐标（x_1, y_1）的点 $P1$ 与坐标（x_2,y_2）的点 $P2$ 的曼哈顿距离为：$|x_1-x_2|+|y_1-y_2|$。

① 二维平面两点 $a(x_1,y_1)$与 $b(x_2,y_2)$间的曼哈顿距离为：

$$d_{12}=|x_1-x_2|+|y_1-y_2|$$

② 两个 n 维向量 $a(x_{11},x_{12},\cdots,x_{1n})$与 $b(x_{21},x_{22},\cdots,x_{2n})$间的曼哈顿距离为：

$$d_{12}=\sum_{k=1}^{n}|x_{1k}-x_{2k}|$$

（2）欧氏距离

欧氏距离是最常见的两点之间或多点之间的距离表示法，又称为欧几里得度量，它定义于欧几里得空间中，如点 $x=(x_1,\cdots,x_n)$ 和 $y=(y_1,\cdots,y_n)$之间的距离为：

$$d(x,y)=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2+\cdots+(x_n-y_n)^2}=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

① 二维平面上两点 $a(x_1,y_1)$与 $b(x_2,y_2)$间的欧氏距离为：

$$d_{12}=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

② 三维空间中两点 $a(x_1,y_1,z_1)$与 $b(x_2,y_2,z_2)$间的欧氏距离为：

$$d_{12}=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$$

③ 两个 n 维向量 $a(x_{11},x_{12},\cdots,x_{1n})$与 $b(x_{21},x_{22},\cdots,x_{2n})$间的欧氏距离为：

$$d_{12}=\sqrt{\sum_{k=1}^{n}(x_{1k}-x_{2k})^2}$$

也可以表示成向量运算的形式：

$$d_{12}=\sqrt{(a-b)(a-b)^T}$$

欧式距离有很多变型，汉明距离（Hamming distance）就是其中一种。两个等长字符串 s1 与 s2 之间的汉明距离定义为将其中一个变为另一个所需要做的最小替换次数。例如，字符串“1111”与“1001”之间的汉明距离为 2。

汉明距离经常应用在信息编码中，比如：为了增强容错性，应使编码间的最小汉明距离尽可能大。

（3）切比雪夫距离

若两个向量或两个点 p、q，其坐标分别为 p_i 和 q_i，则两者之间的切比雪夫距离定义如下：

$$D_{chebyshev}(p,q):=\max_i(|p_i-q_i|)$$

这也等于以下 Lp 度量的极值：$\lim_{k\to\infty}\left(\sum_{i=1}^{n}|p_i-q_i|^k\right)^{1/k}$，因此，切比雪夫距离也称为 L∞度量。

在国际象棋中，国王走一步能够移动到相邻的 8 个方格中的任意一格。那么国王从格子（x_1,y_1）走到格子（x_2,y_2），最少需要的步数总是 max（|x_2–x_1|,|y_2–y_1|）步。

① 二维平面两点 $a(x_1,y_1)$与 $b(x_2,y_2)$间的切比雪夫距离为：

$$d_{12}=\max\left(|x_1-x_2|,|y_1-y_2|\right)$$

② 两个 n 维向量 $a(x_{11},x_{12},\cdots,x_{1n})$与 $b(x_{21},x_{22},\cdots,x_{2n})$间的切比雪夫距离为：

$$d_{12}=\max_i\left(|x_{1i}-x_{2i}|\right)$$

这个公式的另一种等价形式如下：

$$d_{12}=\lim_{k\to\infty}\left(\sum_{i=1}^{n}|x_{1i}-x_{2i}|^k\right)^{1/k}$$

2. 与概率统计相关的距离

当一般意义上的物理距离引申到概率统计中时，就会有一些新的概念，比如：标准化欧氏距离（欧氏距离在概率论上的应用）、马氏距离（由协方差距离演变而来）和巴氏距离（离散或连续概率分布的相似性）。

（1）标准化欧氏距离（Standardized Euclidean distance）

标准化欧氏距离是针对简单欧氏距离的缺点进行改进的一种方案。标准欧氏距离的思路是：既然数据各维分量的分布不一样，则先将各个分量都标准化，从而得到期望、方差相等。其中：

期望 $E(X)$是样本的平均值。

方差（协方差）$E\{[X-E(X)][Y-E(Y)]\}$称为随机变量X与Y的协方差，记为 $Cov(X,Y)$。

假设样本集 X 的数学期望或均值（mean）为 m，标准差（standard deviation，方差开根）为 s，那么 X 的“标准化变量” X^*表示为：（$X-m$）/s，而且标准化变量的数学期望为 0，方差为 1。

即样本集的标准化过程（standardization）用公式描述就是：

$$X^* = \frac{X-m}{s}$$

标准化后的值=（标准化前的值－分量的均值）/分量的标准差

经过简单的推导可以得到两个 n 维向量 $a(x_{11},x_{12},\cdots,x_{1n})$与 $b(x_{21},x_{22},\cdots,x_{2n})$间的标准化欧氏距离的公式为：

$$d_{12} = \sqrt{\sum_{k=1}^{n}\left(\frac{x_{1k}-x_{2k}}{s_k}\right)^2}$$

如果将方差的倒数看成是一个权重，那么这个公式可以看成是一种加权欧氏距离（Weighted Euclidean distance）。

（2）马氏距离（Mahalanobis Distance）

马氏距离是由协方差矩阵演变而来的。有 M 个样本向量 $X_1 \sim X_m$，协方差矩阵记为 S，均值记为向量 μ，则其样本向量 X 到 u 的马氏距离表示为：

$$D(X)=\sqrt{(X-\mu)^T S^{-1}(X-\mu)}$$

向量 X_i 与 X_j 之间的马氏距离定义为：

$$D(X_i,X_j)=\sqrt{(X_i-X_j)^T S^{-1}(X_i-X_j)}$$

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布），则公式就成了欧氏距离：

$$D(X_i,X_j)=\sqrt{(X_i-X_j)^T (X_i-X_j)}$$

若协方差矩阵是对角矩阵，公式变成了标准化欧氏距离。马氏距离与量纲无关，排除变量之间相关性的干扰。

（3）巴氏距离（Bhattacharyya Distance）

在统计中，巴氏距离测量两个离散或连续概率分布的相似性。它与衡量两个统计样本或种群之间重叠量的巴氏系数密切相关。巴氏距离和巴氏系数是以 20 世纪 30 年代的一个统计学家 A．Bhattacharya 命名的。同时，巴氏系数可以用来确定两个样本被认为是相对接近的，它是用来度量两个样本集的可分性。

对离散概率分布 p 和 q 在同一域 X，巴氏距离被定义为：

$$D_B(p,q)=-\ln(BC(p,q))$$

其中：

$$BC(p,q)=\sum_{x\in X}\sqrt{p(x)q(x)}$$

是巴氏系数。

对于连续概率分布，巴氏系数被定义为：

$$BC(p,q)=\int\sqrt{p(x)q(x)}dx$$

在 $0\leqslant BC\leqslant 1$ 和 $0\leqslant D_B\leqslant\infty$这两种情况下，巴氏距离 D_B 并没有服从三角不等式。如果两个样本完全没有重叠，巴氏系数将会等于 0。

各种距离的应用场景简单地概括为如下情况。

① 对于一般意义上的物理距离，则有

- 空间：用欧氏距离；
- 编码差别：用汉明距离。
- 路径：用曼哈顿距离；
- 国际象棋国王：用切比雪夫距离；
- 欧氏距离、曼哈顿距离、切比雪夫距离三种距离的统一形式是闵可夫斯基距离。

② 对于离散数据的概率距离，则有

- 加权：标准化欧氏距离；
- 排除量纲和依存：马氏距离；
- 概率分布的相似性：巴氏距离。

6.1.2 相关系数

由于习惯的原因，我们把两组样本近似线性的数据的距离称为相关系数。相关系数是衡量相似度的主要指标之一。

相关系数属于最重要的数据挖掘的概念之一。目前有两种重要的相关系数：夹角余弦（在两组数据中，可称为皮尔逊积矩相关系数）和杰卡德相似系数。其中，夹角余弦是在 LBS 中应用最普遍的相关系数。

1. 夹角余弦

在二维空间中，向量 $A(x_1,y_1)$与向量 $B(x_2,y_2)$的夹角余弦公式为：

$$\cos\theta=\frac{x_1x_2+y_1y_2}{\sqrt{x_1^{\ 2}+y_1^{\ 2}}\sqrt{x_2^{\ 2}+y_2^{\ 2}}}$$

如果有两组样本数据，两组 n 维样本点 $a(x_{11},x_{12},\cdots,x_{1n})$和 $b(x_{21},x_{22},\cdots,x_{2n})$的夹角余弦为：

$$\cos\theta=\frac{a\cdot b}{|a||b|}$$

类似地，对于两个 n 维样本点 $a(x_{11},x_{12},\cdots,x_{1n})$和 $b(x_{21},x_{22},\cdots,x_{2n})$，可以使用类似

于夹角余弦的概念来衡量它们间的相似程度，即

$$\cos\theta = \frac{\sum_{k=1}^{n} x_{1k} x_{2k}}{\sqrt{\sum_{k=1}^{n} x_{1k}^{\ 2}} \sqrt{\sum_{k=1}^{n} x_{2k}^{\ 2}}}$$

夹角余弦取值范围为[–1,1]。夹角余弦越大，表示两个向量的夹角越小；夹角余弦越小，表示两个向量的夹角越大。当两个向量的方向重合时，夹角余弦取最大值 1；当两个向量的方向完全相反时，夹角余弦取最小值–1。

（1）相关系数的概念

如果将夹角余弦的概念再引申一下，引申到两组数据回归直线的夹角的余弦，则得到皮尔逊积矩相关系数（又称为 PPMCC 或 PCCs，一般简称为相关系数），用于度量两组变量 X 和 Y 之间的相关性（线性相关）。在 LBS 领域，该系数广泛用于度量两个向量（即：两组变量）之间的相关程度。

图 6-1 中的回归直线：$y=f(x)$和 $x=f(y)$。其中，a_x、b_x 与 a_y、b_y 属于线性回归方程的系数。

相关系数是两组数据中心化后夹角 θ 的余弦值，即等于两条回归线 $y=f(x)$和 $x=f(y)$ 夹角的余弦值。

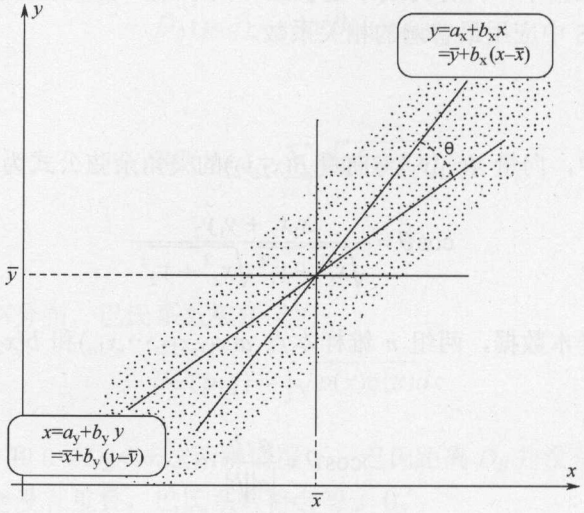


图 6-1　相关系数

具体地说，相关系数等于两组变量之间的协方差和标准差的商：

$$\rho_{X,Y}=\frac{\operatorname{cov}(X,Y)}{\sigma_X\sigma_Y}=\frac{E\left(\left(X-\mu_X\right)\left(Y-\mu_Y\right)\right)}{\sigma_X\sigma_Y}=\frac{E(XY)-E(X)E(Y)}{\sqrt{E(X^2)-E^2(X)}\sqrt{E(Y^2)-E^2(Y)}}$$

相关距离的定义是：$D_{xy}=1-\rho_{XY}$。

以上方程定义了总体相关系数，一般表示成希腊字母 ρ(rho)。基于样本对协方差和方差进行估计时，一般表示成 r：

$$r=\frac{\sum_{i=1}^{n}\left(X_i-\bar{X}\right)\left(Y_i-\bar{Y}\right)}{\sqrt{\sum_{i=1}^{n}\left(X_i-\bar{X}\right)^2}\sqrt{\sum_{i=1}^{n}\left(Y_i-\bar{Y}\right)^2}}$$

一种等价表达式表示成标准分的均值。基于（X_i,Y_i）的样本点，样本的相关系数为：

$$r=\frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{X_i-\bar{X}}{s_X}\right)\left(\frac{Y_i-\bar{Y}}{s_Y}\right)$$

其中，$\frac{X_i-\bar{X}}{s_X}$、$\bar{X}$ 和 s_X 分别是标准分、样本平均值和样本标准差。

（2）相关系数的适用范围

当两个变量的标准差都不为零时，相关系数才有定义，皮尔逊相关系数适用于以下情况。

- 两个变量之间是线性关系，都是连续数据；
- 两个变量的总体是正态分布，或接近正态的单峰分布；
- 两个变量的观测值是成对的，每对观测值之间相互独立。

（3）相关系数的应用

假设有 5 个城市的储蓄分别为 1 亿元、2 亿元、3 亿元、5 亿元和 8 亿元，而这 5 个城市的犯罪率百分比分别为 11%、12%、13%、15% 和 18%。令 x 和 y 分别为包含上述 5 个数据的向量：x=（1,2,3,5,8）和 y=（0.11,0.12,0.13,0.15,0.18）。

利用通常的方法计算两个向量之间的夹角，未中心化的相关系数如下：

$$\cos\theta = \frac{x \cdot y}{\|x\|\|y\|} = \frac{2.93}{\sqrt{103}\sqrt{0.0983}} = 0.920814711$$

将数据中心化，即通过 $E(x) = 3.8$ 移动 x 和通过 $E(y) = 0.138$ 移动 y ，得到：

$$x=(-2.8, -1.8, -0.8, 1.2, 4.2)$$

$$y=(-0.028, -0.018, -0.008, 0.012, 0.042)$$

从而有

$$\cos\theta = \frac{x \cdot y}{\|x\|\|y\|} = \frac{0.308}{\sqrt{30.8}\sqrt{0.00308}} = 1 = \rho_{xy}$$

2. 杰卡德相似系数（Jaccard similarity coefficient）

杰卡德相似系数是衡量两个集合相似度的一种指标。具体地说，两个集合 A 和 B 的交集元素在 A、B 的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号 $J(A,B)$表示为：

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

杰卡德距离可用如下公式表示：

$$J_\delta(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

杰卡德相似系数与杰卡德距离的应用情况如下。

假设样本 A 与样本 B 是两个 n 维向量，而且所有维度的取值都是 0 或 1，例如：A(0111)和 B(1011)。我们将样本看成是一个集合，1 表示集合包含该元素，0 表示集合不包含该元素。

- M_{11}：样本 A 与 B 都是 1 的维度的个数。
- M_{01}：样本 A 是 0，样本 B 是 1 的维度的个数。
- M_{10}：样本 A 是 1，样本 B 是 0 的维度的个数。

- M_{00}：样本 A 与 B 都是 0 的维度的个数。

根据杰卡德相似系数和杰卡德距离的相关定义，样本 A 与 B 的杰卡德相似系数 J 可以表示为：

$$J=\frac{M_{11}}{M_{01}+M_{10}+M_{11}}$$

其中，$M_{11}+M_{01}+M_{10}$ 可理解为 A 与 B 的并集的元素个数，而 M_{11} 是 A 与 B 的交集的元素个数，则样本 A 与 B 的杰卡德距离表示为 J'：

$$J'=\frac{M_{01}+M_{10}}{M_{01}+M_{10}+M_{11}}$$

6.2 数据分类

分类问题是数据挖掘的核心，它分为两种：聚类（静态分类）和机器学习（动态分类，通常是一种利用训练的预测问题）。聚类问题作为一种定量方法，从数据分析的角度给出了一种分类工具。机器学习则需要用训练数据进行学习，从而进行分类。机器学习经常归为搜索问题，即对非常大的假设空间进行搜索，以确定能最佳拟合观察到的数据和学习器已有知识的假设。

6.2.1 聚类

聚类属于一种无指导的分类问题。比如：给出一堆数据点，将它们分成一堆一堆的。典型的聚类如图 6-2 所示。

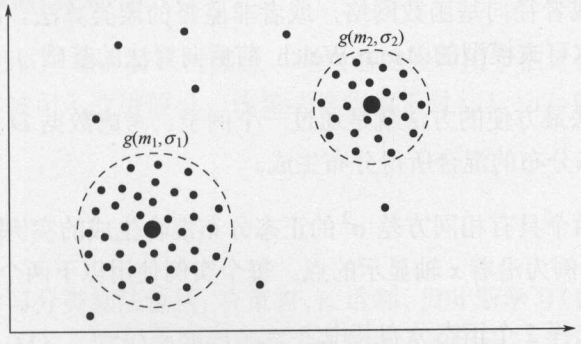


图 6-2　典型聚类

聚类的算法有很多，但基本上都是通过密度，或类似密度的做法来进行聚类。常用的算法有如下几种。

1. DBSACN 算法

如果只是得到数据中符合某稠密度的数据，则只需要遍历所有的点，如果点的周围某个范围内有数据，则此点保留，否则此点被删除。这种算法被称为 DBSACN 算法。

该算法在 LBS 中应用最普遍，比如：通过兴趣点来找某个商业区，用于点云处理等。

2. K-Means 算法

K-Means（K-均值聚类）算法是一个聚类算法，把 n 的对象根据它们的属性分为 k 个分割（$k<n$）。它假设对象属性来自空间向量，并且目标是使“各个群组内部的均方误差总和/样本数量”小于设定值。

3. EM 算法

在许多实际的学习问题框架中，相关实例特征中只有一部分可观察到。例如，在训练贝叶斯置信网时，可能网络变量中只有一个子集能在数据中观察到。已有许多方法被提出用来处理存在未观察到变量时的问题。基本的思路是：若某些变量有时能观察到，有时不能，那么可以用观察到的实例去预测未观察到的。EM 算法（Expectation–Maximization，最大期望算法）就是这样一种存在隐含变量的情况下应该使用的学习方法。EM 算法可用于变量的值从来没有被直接观察到的情形，只要这些没有被直接观察到的隐含变量所遵循的概率分布是已知的。EM 算法已被用于训练贝叶斯置信网，或者径向基函数网络，或者非监督的聚类算法。而且它是用于学习部分可观测的马尔可夫模型的 Baum-Welch 前后向算法的基础。

介绍 EM 算法最方便的方法就是通过一个例子。考虑数据 D 是一个实例集合，它由 k 个不同正态分布的混合所得分布生成。

图 6-3 是由两个具有相同方差 σ^2 的正态分布混合生成的实例。也就是说，实例个数 k=2，而且实例为沿着 x 轴显示的点。每个实例使用以下两个步骤形成。

首先，随机选择 *2* 个正态分布其中之一。

其次，随机变量 x_i 按照此正态分布生成，其中 x_i 表示第 i 个实例，每个实例都可以表示成<x_i，z_{i1} ，z_{i2} >，这里的 z_{ij} 表示第 j 个正态分布的第 i 个实例。

这一过程不断重复，生成一组要处理的数据。为简单起见，考虑一个简单的情形，即基于均匀的概率选择每个正态分布，并且 k 个正态分布有相同的方差 σ^2，且 σ^2 已知。学习任务是输出一个假设 h=<μ1…μk>，它描述了 k 个分布中每一个分布的均值。我们希望对这些均值找到一个极大似然假设，即一个使 $P(D|h)$最大化的假设 h。

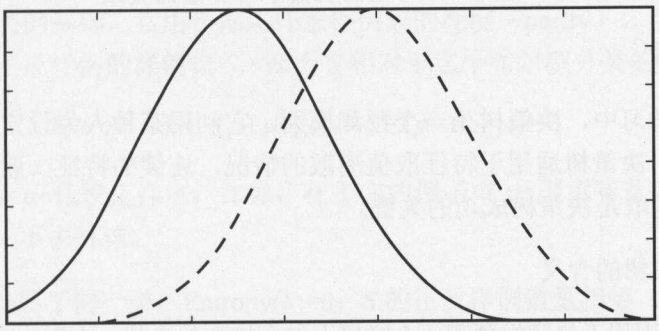


图 6-3　EM 算法实例

在这种情况下，EM 算法的具体步骤如下。

步骤 1：将假设初始化为 h=<μ1，μ2>，其中 $\mu 1$ 和 $\mu 2$ 为任意的初始值。计算每个隐藏变量 z_{ij} 的期望值 $E[z_{ij}]$。

步骤 2：计算一个新的极大似然假设 h'=<μ1′，μ2′>，假定由每个隐藏变量 z_{ij} 所取的值为步骤 1 中得到的期望值 $E[z_{ij}]$，然后将假设 h=<μ1，μ2>替换为新的假设 h'=<μ1′，μ2′>，最后返回步骤 1 循环。

上面的步骤描述了 EM 算法的要点，即当前的假设用于估计未知的变量，而这些变量的期望值再被用于改进假设。该算法将收敛于对<μ1，μ2>的一个局部最大似然假设。

6.2.2　机器学习

常用的机器学习分类算法包括：决策树、K 近邻、贝叶斯学习（Bayesian classifier）和支持向量机（SVM），它们之间的关系大致如图 6-4 所示。

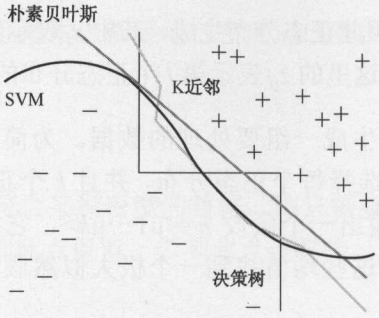


图 6-4　常见的机器学习算法的关系

1. 决策树

在机器学习中，决策树是一个预测模型，它利用某种人为设定的特征规则来做出某种决策。决策树适用于特征取值离散的情况，连续的特征一般也要处理成离散的。特征的选取是决策树成功的关键。

（1）决策树的含义

假设通过下周天气预报预测什么时候人们会打高尔夫球，人们决定是否打球的原因最主要取决于天气情况，而天气状况有晴、云和雨；气温用华氏温度表示；相对湿度用百分比；有无风。如此便可以构造出一棵决策树（如图 6-5 所示，这个分类决策根据天气决定当天是否适合打网球）。

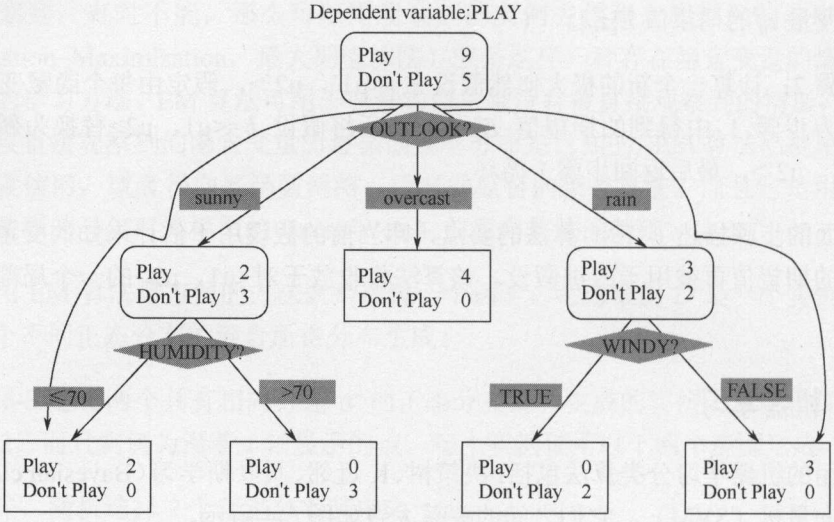


图 6-5　决策树

从图 6-5 可以看出，决策树在本质上和多维空间的 K-d 树（见第 5 章中的 K-d 树索引部分，5.1.2 节）类似。因为决策树也是将空间用超平面进行划分的一种方法，每次分割的时候，都将当前的空间一分为二（可超过二，比如图 6-5 的根结点的子结点就是 3）。这样使得每一个叶子结点都是在空间中一个不相交的区域，在进行决策的时候，会根据输入样本每一维特征的值，一步一步往下，最后使样本落入 N 个区域中的一个（假设有 N 个叶子结点）。

（2）样例集的纯度

熵是进行分类的基础，它用于描述任意样例集的纯度（purity）。给定包含关于某个目标概念的正反样例的样例集 S，那么 S 相对于这个布尔型分类的熵的含义为：

$$Entropy(s) \equiv -(p+)\log_2(\rho+) - (p-)\log_2(\rho-)$$

上述公式中，p+代表正样例，比如，在上文的例子中 p+则意味着去打网球，而 p-则代表反样例，不去打球。

S 的所有成员属于同一类，Entropy(S)=0；S 的正反样例数量相等（$p+ = p-$），Entropy(S)=1；S 的正反样例数量不等，熵介于 0 和 1 之间，如图 6-6 所示。

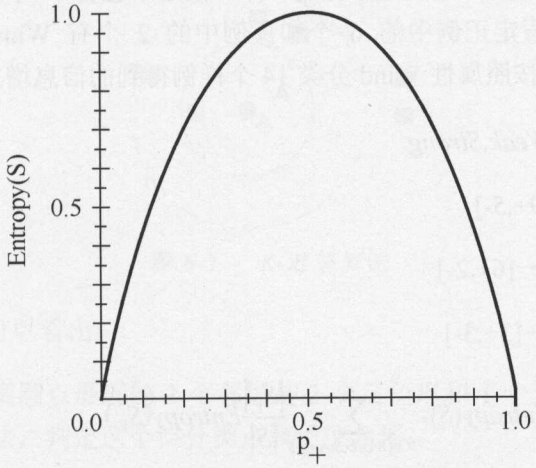


图 6-6　熵

信息论中对熵的一种解释是：熵确定了要编码集合 S 中任意成员的分类所需要的最少二进制位数。

（3）信息增益

信息增益（Information Gain）是用来衡量给定的属性区分训练样例的能力，信息增益本质上是衡量给定属性能否更完美地没有误差地区分开训练样例。用数学的语言说，一个属性的信息增益就是由于使用这个属性分割样例而导致的期望熵降低。更精确地讲，一个属性 A 相对样例集合 S 的信息增益 $Gain$（S，A）被定义为：

$$Gain(S,A)=Entropy(S)-\sum_{v\in V(A)}\frac{|S_v|}{|S|}Entropy(S_v)$$

其中，V（A）是属性 A 的值域。

S 是样本集合。

S_v 是 S 中在属性 A 上值等于 v 的样本集合。

当对 S 的一个任意成员的目标值编码时，$Gain$（S，A）的值是在知道属性 A 的值后可以节省的二进制位数。

举例说明，假定 S 是一套有关天气的训练样例，描述它的属性包括可能是具有 Weak 和 Strong 两个值的 Wind。像前面一样，假定 S 包含 14 个样例，[9+，5–]。在这 14 个样例中，假定正例中的 6 个和反例中的 2 个有 Wind =Weak，其他的有 Wind=Strong。由于按照属性 Wind 分类 14 个样例得到的信息增益可以计算如下。

$Values(Wind)=Weak,Strong$

$S=[9+,5-]$

$S_{Weak}\leftarrow[6+,2-]$

$S_{Strong}\leftarrow[3+,3-]$

$$Gain(S,Wind)=Entropy(S)-\sum_{V\in\{Weak,Strong\}}\frac{|S_V|}{|S|}Entropy(S_V)$$

$$=Entropy(S)-(8/14)\ Entropy(S_{Weak})-(6/14)\ Entropy(S_{Strong})$$

$$=0.940-(8/14)*0.811-(6/14)*1.00$$

$$=0.048$$

再按照湿度进行信息增益的计算，最终 humidity 这种分类的信息增益 0.151>wind 增益的 0.048。即在星期六上午是否适合打网球的问题决策中，采取 humidity 较 wind 为分类属性更佳。这就建立了一个更好的决策树。

2. *K*-近邻算法

K-近邻算法（K-Nearest Neighbor algorithm，简称 KNN 算法）是一种利用周围 *K* 个邻居的算法来对某个实例分类的方法。*K*-近邻是一个最基本的基于实例的学习方法。*K*-近邻算法往往可以先建立空间索引，之后再对每个区域进行近邻算法处理。

这个算法是假定所有的实例对应 *n* 维欧氏空间$\Re n$ 中的点。一个实例的最近邻是根据标准欧氏距离定义的。

（1）*K*-近邻算法的实例

如图 6-7 所示，有两类不同的样本数据，分别用正方形和三角形表示，而正中间的那个圆所标示的数据则是待分类的数据。也就是说，现在如何知道中间那个圆是正方形还是三角形？

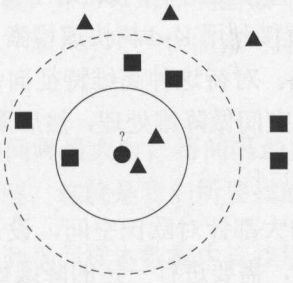


图 6-7　*K*-近邻算法

从图 6-7 中，可以看出：

如果 *K*=3，距离圆点最近的 3 个邻居是 2 个三角形和 1 个正方形，少数从属多数，基于统计的方法，判定这个待分类点属于三角形。

如果 *K*=6，距离圆点最近的 6 个邻居是 2 个三角形和 4 个正方形，还是少数从属多数，基于统计的方法，判定这个待分类点属于正方形。

这说明，当无法判定当前待分类点是从属于已知分类中的哪一类时，可以依据统计学的理论看它所处的位置特征，衡量它与周围邻居的距离，而把它归为到距离

更小的那一类。这就是 *K*-近邻算法的核心思想。在实际应用中，*K* 值一般取一个比较小的数值，例如，采用交叉验证法（简单地说，就是一部分样本作为训练集，一部分作为测试集）来选择最优的 *K* 值。

对前面的 *K*-近邻算法进行简单修改后，它就可用于逼近连续值的目标函数。为了实现这一点，我们让算法计算 *K* 个最接近样例的平均值，而不是计算其中最普遍的值。

对 *K*-近邻算法的另一个显而易见的改进是对 *K* 个近邻的贡献加权，根据它们相对于查询点 *xq* 的距离，将较大的权值赋给较近的近邻。例如，在逼近离散目标函数的算法中，我们可以根据每个近邻与 *xq* 的距离平方的倒数加权这个近邻的“选举权”。

按距离加权的 *K*-近邻算法是一种非常有效的归纳推理方法。它对训练数据中的噪声有很好的鲁棒性，而且当给定足够大的训练集合时，它也非常有效。注意，通过取 *K* 个近邻的加权平均，可以消除孤立的噪声样例的影响。

（2）*K*-近邻算法在高维空间的应用

为了介绍方便，以上讨论的是二维或三维情形，这在 LBS 的大部分应用中已经足够了。但在基于内容的图像检索应用中，如 SIFT 特征矢量 128 维、SURF 特征矢量 64 维，其维度都比较大，直接利用 K-d 树快速检索（维数不超过 20）的性能急剧下降，几乎接近贪婪线性扫描。对待这种高维特征向量的距离索引问题，目前常用的解决办法是首先对高维特征空间做降维处理，然后采用包括四叉树、K-d 树、R 树族等在内的主流多维索引结构。

目前主流的多维索引结构大都针对欧氏空间，设计需要利用到欧氏空间的几何性质，所以针对多维特征数据，需要进行一定的降维处理。

应用 *K*-近邻算法的一个实践问题是如何建立高效的索引。因为这个算法接收到一个新的查询后，才会进行大规模的数据处理，所以，处理每个新查询可能需要大量的计算。目前已经开发了很多方法用来对存储的训练样例进行索引，以便在增加一定存储开销的情况下更高效地确定最近邻。这些索引方法在 5.1.2 节中有详细描述。

3. 支持向量机的分类器

支持向量机因其英文名为 Support Vector Machine，故一般简称为 SVM，其基本模型定义为特征空间上间隔最大的线性分类器，其学习策略便是间隔最大化，最终用一个超平面把样本数据切分成两个半平面，这个超平面就是 SVM。SVM 可应用在很多

领域，如文本分类、图像分类、生物序列分析和生物数据挖掘，以及手写字符识别。

举例来说，如图 6-8 所示，现在有一个二维平面，平面上有两种不同的数据，分别用圈和叉表示。由于这些数据是线性可分的，所以可以用一条直线将这两类数据分开，这条直线就相当于一个超平面，超平面一边的数据点所对应的 y 全是负，另一边所对应的 y 全是正。

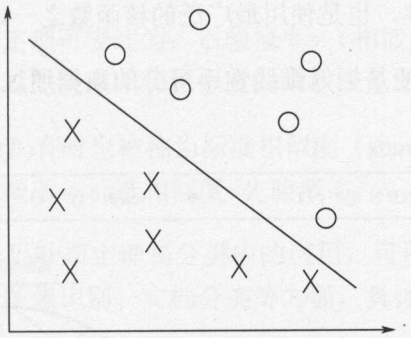


图 6-8　支持向量机的分类器

这个超平面就是线性向量机，并可以用分类函数 $f(x)=\omega^{T}x+b$ 表示，当 $f(x)$ 等于 0 时，x 便是位于超平面上的点，而 $f(x)$大于 0 的点对应 y 为正的数据点，$f(x)$小于 0 的点对应 y 为负的点。

当超平面所对应的样本的间隔最大时，将间隔最大作为拉格朗日函数的约束条件，可以得到平面的最优化的解，这就是我们所要找的最佳的分类平面。

向量机可以很容易地应用到平面样本数据中查找用于分类的超平面，公式如下：

$$f(x)=\sum_{i=1}^{n}\alpha_i y_i(x_i,x)+b$$

由上面的公式可以看出，SVM 与神经网络有类似之处。SVM 是一个超平面，而神经网络也是 n 维实例空间中的超平面决策面。

对于二维线性不可分的两类样本点可以利用高斯核函数，映射到高维空间后，再找到切分数据的超平面。现在我们的分类函数如下：

$$\sum_{i=1}^{n}\alpha_i y_i k(x_i,x)+b$$

其中，高斯核函数 $k(x_1,x_2)=\exp\left(-\frac{\|x_1-x_2\|^2}{2\sigma^2}\right)$。

如果 σ 取值很大，高次特征上的权重实际上衰减得非常快，所以实际上相当于一个低维的子空间；反之，如果 σ 取值很小，则可以将任意数据映射为线性可分，但随之而来的可能是非常严重的过拟合问题。总的来说，通过调控参数 σ，高斯核实际上具有相当高的灵活性，也是使用最广泛的核函数之一。

图 6-9 所示的例子便是把低维线性不可分的数据通过高斯核函数映射到了高维空间。

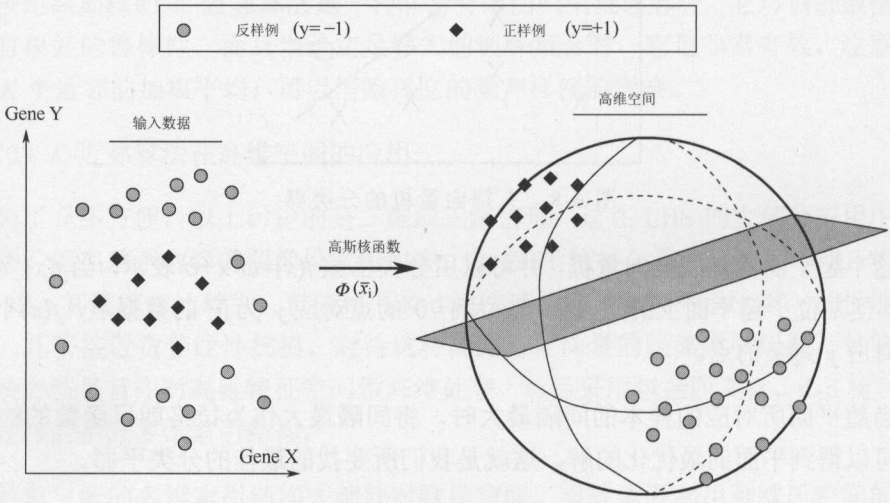


图 6-9　利用高斯核函数映射到高维空间

4. 贝叶斯学习

贝叶斯定理是关于随机事件 A 和 B 的条件概率和边缘概率的一则定理。

$$P(A\mid B)=\frac{P(B\mid A)P(A)}{P(B)}$$

其中，$P(A|B)$是在 B 发生的情况下 A 发生的可能性。在贝叶斯定理中，每个名词都有约定俗成的名称。

$P(A)$是 A 的先验概率或边缘概率。之所以称为“先验”，是因为它不考虑 B 方面的任何因素。

$P(A|B)$是已知 B 发生后 A 的条件概率（也就是说，先有 B，然后才有 A），也由于得自 B 的取值而被称为 A 的后验概率。

$P(B|A)$是已知 A 发生后 B 的条件概率（也就是说，先有 A，然后才有 B），也由于得自 A 的取值而被称为 B 的后验概率。

$P(B)$是 B 的先验概率或边缘概率，也称之为标准化常量（normalized constant）。

按这些术语，贝叶斯定理可表述为：后验概率=（相似度×先验概率）/标准化常量。也就是说，后验概率与先验概率和相似度的乘积成正比。

另外，比例 $P(B|A)/P(B)$有时也被称为标准相似度（standardised likelihood），贝叶斯定理可表述为：后验概率 = 标准相似度×先验概率。

贝叶斯学习算法就是贝叶斯定理在分类中的应用，可用于分词、拼写检查、垃圾邮件过滤、文章推荐、图像识别、文档分类等方面，具体介绍如下。

（1）分词

分词问题描述为：给定一个句子（字串），如：南京市长江大桥。如何对这个句子进行分词（词串）才是最靠谱的？例如：

① 南京市/长江大桥

② 南京/市长/江大桥

对于这两个分词，到底哪个更准确呢？

为了降低概率的依赖，我们假设句子中一个词的出现概率只依赖于它前面有限的 k 个词（k 一般不超过 3，如果只依赖于前面的一个词，就是二元语言模型），这就是所谓的“有限地平线”假设。于是，概率改写成：$P(W2|W1) = P(W1) * P(W2|W1) * P(W3|W2) * P(W4|W3)\cdots$（假设每个词只依赖于它前面的一个词）。

对上面提到的例子“南京市长江大桥”，如果按照自左到右的贪婪方法分词，结果就成了“南京市长/江大桥”。

如果按照贝叶斯分词（假设使用二元语法），由于“南京市长”和“江大桥”在语料库中一起出现的频率为 0，这个整句的概率便会被判定为 0，从而使“南京市/长江大桥”这一分词方式胜出。

（2）拼写检查或纠正

如果我们看到用户输入了一个不在字典中的单词，则需要去猜测："这个家伙到底真正想输入的单词是什么？"，即 P（我们猜测他想输入的单词 | 他实际输入的单词），即 $P(h|D)$。

运用一次贝叶斯公式得到：

$$P(h|D)=P(h) * P(D|h)/P(D)$$

比如，用户输入了 wer，但是因为字典中没有这个词，所以，这可能是一个需要校正的词。用户实际上想输入 we 的可能性大小取决于 we 本身在词汇表中被使用的频繁程度大小（先验概率）和想输入 we 却输成 wer 的可能性大小（似然）的乘积。

剩下的工作就很简单了，对我们猜测为可能的每个单词计算 $P(h)* P(D|h)$ 值，然后取最大的值，得到的就是最靠谱的猜测。

（3）垃圾邮件过滤（或用户喜欢的文章推荐）

给定一封邮件，判定它是否属于垃圾邮件。按照先例，我们还是用 D 来表示这封邮件，注意，D 由 N 个单词组成。我们用 $h+$来表示垃圾邮件，$h-$表示正常邮件。问题可以形式化地描述为：

$$P(h+|D)=P(h+)*P(D|h+)/P(D)$$

$$P(h-|D)=P(h-)*\mathrm{P}(D|h-)/P(D)$$

因为 D 中含有 N 个单词 $d1$，$d2$，$d3$，…，所以，$P(D|h+) = P(d1,d2,\cdots,dn|h+)$。

我们将 $P(d1,d2,\cdots,dn|h+)$扩展为：$P(d1|h+) * P(d2|d1,h+) * P(d3|d2,d1,h+) *\cdots$

这里引入一个更激进的假设，假设 di 与 $di-1$ 是完全条件无关的，于是式子就简化为 $P(d1|h+) * P(d2|h+) * P(d3|h+)*\cdots$。这个假设就是条件独立假设。引入条件独立假设后，贝叶斯方法就变成了朴素贝叶斯方法。这时，计算 $P(d1|h+) * P(d2|h+) * P(d3|h+) *\cdots$，只要统计 di 这个单词在垃圾邮件中出现的频率即可。

（4）图像识别

图像识别的核心理念可以描述成通过合成来分析。如图 6-10 所示，首先是视觉系统提取图形的边角特征，然后使用这些特征自底向上地激活高层的抽象概念（比

如是 E 、F 还是等号）。最后使用一个自顶向下的验证来比较到底哪个概念最佳地解释了观察到的图像。

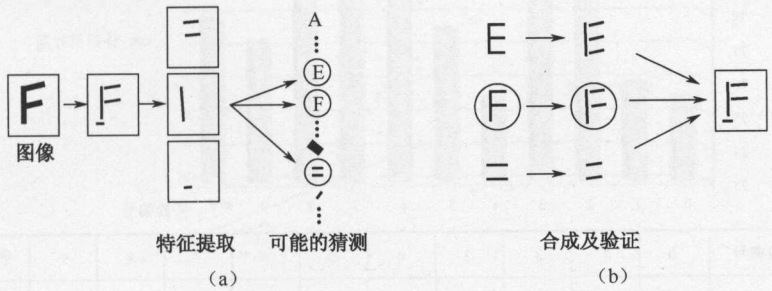


图 6-10　贝叶斯图像识别

（5）文档分类

在经过文档预处理、特征词的选取之后，下面用朴素贝叶斯算法对文档集做分类，看看此算法的效果。

实验结果为：取所有的词共 76554 个作为特征词：10 次交叉验证实验平均准确率为 78.19%，用时 23 分，准确率范围为 75.65%～80.47%。如果取出现次数大于或等于 4 次的词，则共计 30097 个作为特征词：10 次交叉验证实验平均准确率为 77.91%，用时 22 分，准确率范围为 75.51%～80.26%。文档分类的结果如图 6-11 所示。

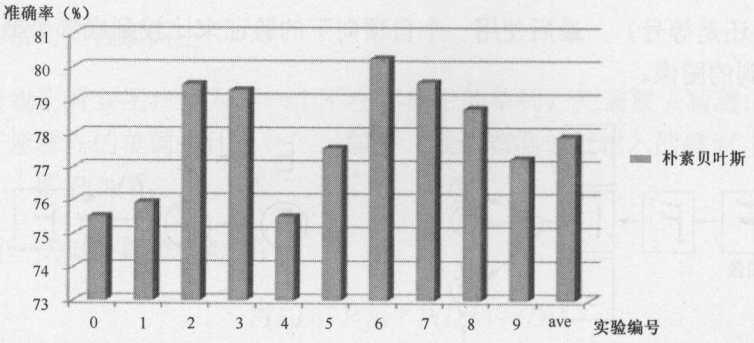
可以看出，朴素贝叶斯算法不必去除出现次数很低的词，因为出现次数很低的词的 IDF 比较大，去除后，分类准确率下降，而计算时间并没有显著减少。

为了进一步提高此算法的分类准确率，可以考虑以下情况。

① 优化特征词的选取策略。

② 改进多项式模型的类条件概率的计算公式，改进为类条件概率 $P(tk|c)$=（类 c 下单词 tk 在各个文档中出现过的次数之和+0.001）/（类 c 下单词总数+训练样本中不重复的特征词总数），分子中当单词 tk 没有出现时，只加 0.001，这样更加精确地描述词的统计分布规律。

改进后的实验结果为：第 6 次分组实验的准确率提高到 84.79%，第 7 次分组实验的准确率达到 85.24%，平均准确率由 77.91%提高到 82.23%，优化效果还是很明显的。



实验编号	0	1	2	3	4	5	6	7	8	9	平均
准确率（%）	75.51	75.93	79.48	79.30	75.51	77.57	80.26	79.54	78.75	77.25	77.91

图 6-11　文档分类

6.3　图像识别

图像识别有两种常用的方法：RANSAC 算法和 HOUGH 变换。

6.3.1　RANSAC 算法

RANSAC（random sample consensus）算法是利用调节直线的参数进行边线检测的一种方法。

RANSAC 算法在本质上是对最小二乘法的一种优化。

1. 最小二乘法的缺点

最小二乘法是线性回归的代名词，但只适合于误差较小的情况。试想一下：假设需要从一个噪声较大的数据集中提取模型（假设只有 20%的数据是符合模型的），最小二乘法就显得力不从心了。如图 6-12 所示，肉眼可以很轻易地看到一条直线，但最小二乘法却找错了，其原因在于当特征点很少时，不应该被视为有效。

RANSAC 算法就是针对最小二乘法的优化，RANSAC 算法在足够多的点被归类为假设的局内点时才认为是有效假设。

具体地说，RANSAC 算法的输入是一组观测数据（往往含有较大的噪声或无效点），一个用于解释观测数据的参数化模型以及一些可信的参数。RANSAC 通过反复选择数据中的一组随机子集来达成目标，比如：随机选择两个点来识别直线。被

选取的子集被假设为局内点，并用下述方法进行验证。

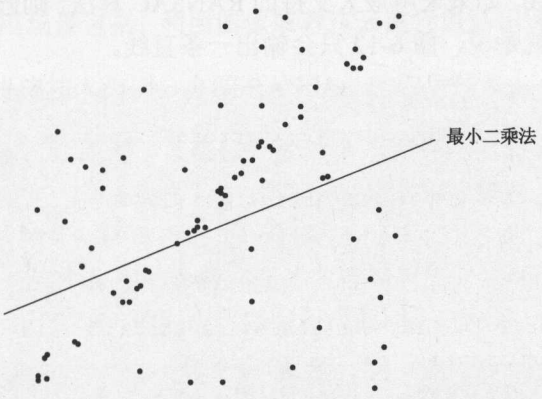


图 6-12　最小二乘法

有一个线性模型适用于假设的局内点，即所有的未知参数都能从假设的局内点计算得出。

如果有足够多的点被归类为假设的局内点，那么估计的模型就足够合理，否则选取下一个随机子集，换用下一个线性模型。

重复执行上述步骤，直到得到最优的线性模型。

RANSAC 算法的过程可参考图 6-13。

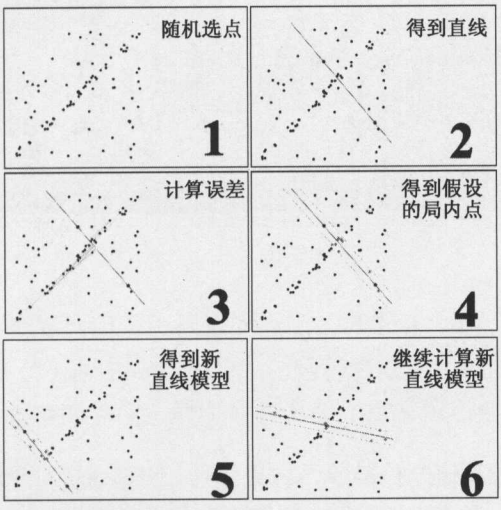


图 6-13　RANSAC 算法

图 6-13 中的图 6 因为没有足够的假设局内点，所以将继续执行，直到找到足够的假设局内点的直线。如果采用最大支持的 RANSAC 算法，则假设的局内点最多时，才会输出假设。也就是说，图 6-12 只会输出一条直线。

```
    template<class T, class S>
    double Ransac<T,S>::compute(std::vector<S> &parameters,

ParameterEsitmator<T,S> *paramEstimator ,
                                                    std::vector<T> &data,
                                                    int numForEstimate)
    {
        std::vector<T *> leastSquaresEstimateData;
        int numDataObjects = data.size();
        int numVotesForBest = -1;
        int *arr = new int[numForEstimate];
    // numForEstimate 表示拟合模型所需要的最少点数，对本例的直线来说，该值为 2
        short *curVotes = new short[numDataObjects];
        short *bestVotes = new short[numDataObjects];

        if(numDataObjects < numForEstimate)
            return 0;
           // 计算所有可能的直线，寻找其中误差最小的解。对 100 点的直线拟合来说，大约需
    //要 100*99*0.5=4950 次运算，复杂度无疑是庞大的。一般采用随机选取子集的方式
        computeAllChoices(paramEstimator,data,numForEstimate,
                                        bestVotes, curVotes,
numVotesForBest, 0, data.size(), numForEstimate, 0, arr);

           //计算最小二乘估计
        for(int j=0; j<numDataObjects; j++) {
            if(bestVotes[j])
             leastSquaresEstimateData.push_back(&(data[j]));
        }
           // 对局内点再次用最小二乘法拟合出模型
        paramEstimator->leastSquaresEstimate(leastSquaresEstimateData,param
eters);

        delete [] arr;
        delete [] bestVotes;
        delete [] curVotes;

        return
(double)leastSquaresEstimateData.size()/(double)numDataObjects;
    }
```

2. RANSAC 的应用场景

RANSAC 的应用场景包括：利用图像或者点云进行道路识别；图片的拼接技术。

下面以图片的拼接技术为例，介绍 RANSAC 的应用场景。

由于镜头的限制，往往需要多张照片才能拍下那种巨幅的风景。在进行多幅图像的合成时，事先会在待合成的图片中提取一些关键的特征点。计算机视觉的研究表明，不同视角下的物体往往可以通过一个透视矩阵（3×3 或 2×2）的变换得到。RANSAC 被用于拟合这个模型的参数（矩阵各行列的值），从而得到不同的照片具有相同的像素最多，由此便可识别出不同照片中的同一物体，可参考图 6-14。

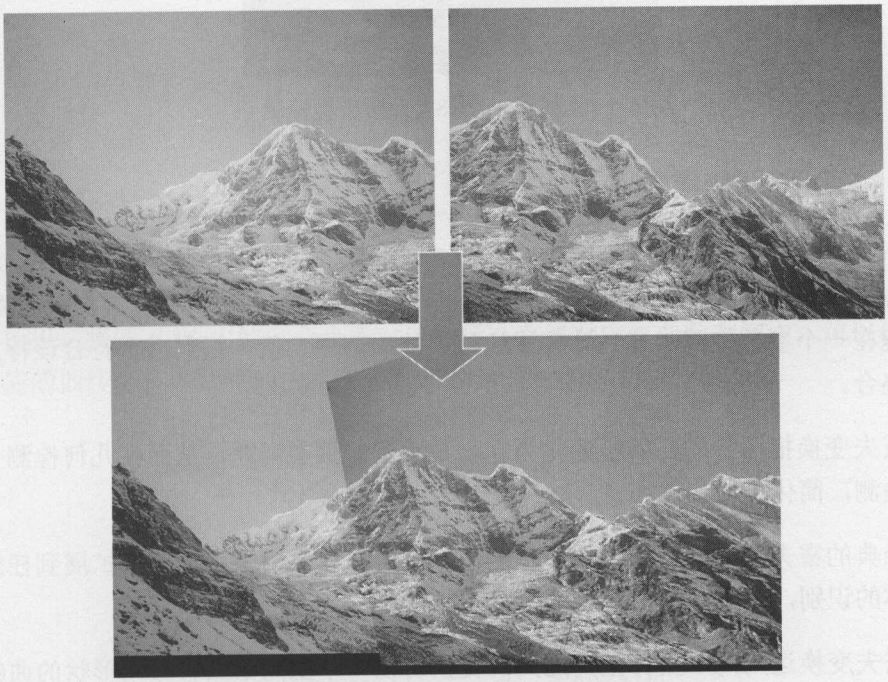


图 6-14　图片拼接

如果把这种算法的思想用于不同激光点云的匹配，则这个算法又可以被称为迭代就近点法（Iterative Closest Point，ICP），这种算法利用激光点云重心位置坐标对两个激光点云进行空间多次移位，从而使两个激光点云可以匹配。

此外，RANSAC 还可以用于图像搜索时的纠错与物体识别定位。图 6-15 中，RANSAC 有效地排除了错误的边线，并识别正确的书本模型，然后用线框标注出来。

图 6-15　利用 RANSAC 算法排除错误

6.3.2　HOUGH 变换

HOUGH 变换（Hough Transform）又称为霍夫变换，是图像处理中的一种特征提取技术，一般用于几何形状检测。它通过一种投票算法检测具有特定形状的物体。该过程在一个参数空间中通过计算累计结果的局部最大值，得到一个符合该特定形状的集合。

霍夫变换把图片的几何检测化为在参数空间的聚类问题，从而使几何检测变为聚类检测，简化了检测过程。

经典的霍夫变换应用示例用于检测图像中的直线，后来霍夫变换扩展到任意形状物体的识别，多为圆和椭圆。

霍夫变换运用两个坐标空间之间的变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。

我们知道，一条直线在直角坐标系下可以用 $y=kx+b$ 表示，霍夫变换的主要思想是将该方程的参数和变量交换用 x、y 作为已知量，k、b 作为变量坐标，所以直角坐标系下的直线 $y=kx+b$ 在参数空间表示为点（k,b），而一个点（x_1,y_1）在参数坐标系下表示为无数条直线 $y_1=kx_1+b$，其中，（k,b）是该直线上的任意点。为了计算方便，我们将参数空间的坐标表示为极坐标下的 γ 和 θ。因为在直角坐标系下，同一条直线上的点

对应的参数空间的（γ,θ）是相同的，因此，可以先将图片进行边缘检测，然后对图像中每一个非零像素点在参数坐标下变换为一条曲线，那么在图像中属于同一条直线的点便在参数空间形成多条曲线并内交于一点。因此，可用该原理进行直线检测。

以图 6-16 中的一条直线为例，有参数（γ,θ）=（69.641,30°）。对于图 6-16 中的任一点（x,y），由于都可以对应无数条直线，所以可以在参数空间形成一条曲线。所有属于同一条直线上的点会在参数空间由三条曲线交于一点，该点即为对应直线的参数。

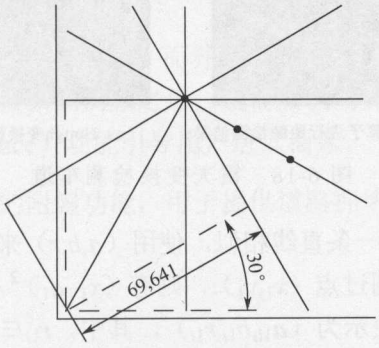


图 6-16　HOUGH 变换

所以，将图 6-16 中的三个点所对应的所有直线对应在参数空间中，可以得到三条对应的曲线。在参数空间的霍夫变换统计结果如图 6-17 所示。

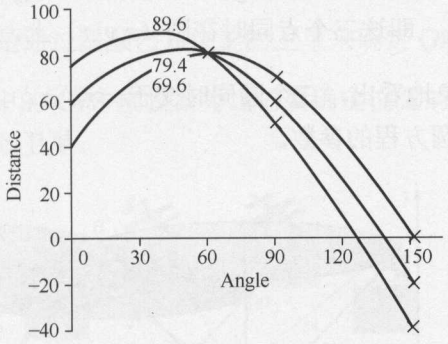
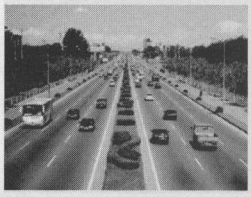


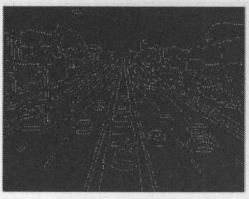
图 6-17　霍夫变换统计结果

霍夫变换检测的结果如图 6-18 所示。

HOUGH 变换检测圆的方法与直线的检测类似，也是对原来的点进行坐标变换，将图像空间转换到参数空间来实现。



（a）原图像



（b）Canny算子进行边缘检测结果

（c）Hough变换检测直线

图 6-18　霍夫变换检测车道

与使用（r,θ）来表示一条直线相似，使用（a,b,r）来确定一个圆心为（a,b）、半径为 r 的圆。由于某个圆过点（x_1,y_1），则有 $(x_1-a_1)^2+(y_1-b_1)^2=r_1^2$ 。那么过点（x_1,y_1）的所有圆可以表示为（a_{1i},b_{1i},r_{1i}），其中，$r_{1i}\in$（0，无穷），每一个 i 值都对应一个不同的圆，（a_{1i},b_{1i},r_{1i}）表示无穷多个过点（x_1,y_1）的圆。

那么，过点（x_1,y_1）的所有圆可以表示为（a_{1i},b_{1i},r_{1i}），过点（x_2,y_2）的所有圆可以表示为（a_{2i},b_{2i},r_{2i}），过点（x_3,y_3）的所有圆可以表示为（a_{3i},b_{3i},r_{3i}），如果这三个点在同一个圆上，那么存在一个值（x,y,z），使得 $x=a_{1k}=a_{2k}=a_{3k}$ 且 $y=b_{1k}=b_{2k}=b_{3k}$ 且 $z=r_{1k}=r_{2k}=r_{3k}$，即这三个点同时在圆（x,y,z）上。

从图 6-19 可以形象地看出，三个圆同时交于一点 A，点 A 所代表的参数就是 HOUGH 变换要检测的圆方程的参数。

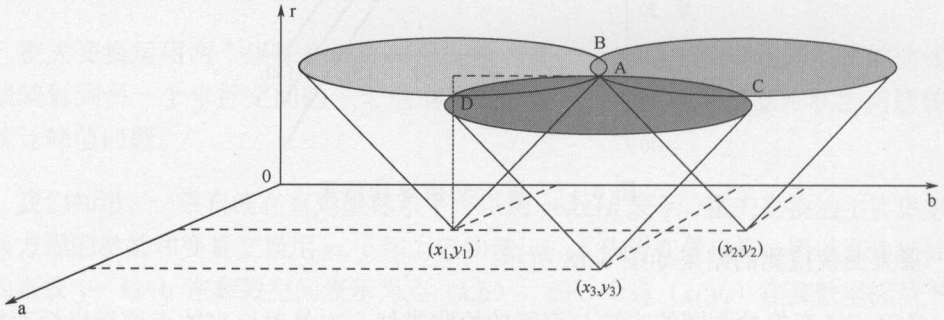


图 6-19　霍夫变换检测圆

第 7 章

导航

导航是 LBS 应用中的基本功能之一，导航功能分为四个部分：定位、算路、路径引导和 TMC（实时交通信息）。

定位是所有 LBS 应用的必备功能，而算路和引导也是大部分 LBS 应用的基础。比如：美团的 O2O 功能，搜索到商家之后，就需要利用算路来找到距离商家最近的距离，并利用引导（Guidance）功能引导用户进入商家。

TMC 功能是在算路时的附属功能，用于提供道路拥堵与否的信息。

7.1 定位

在定位导航技术中，目前精度最高、应用最广泛的是 GPS。由于定位的频率往往在 1 秒钟以上，而这段时间里，手机或者汽车总是运动的，为了得到当前的定位，需要利用卡尔曼滤波的方法，利用历史定位来预测现在的定位。

1. GPS 定位数学模型

GPS 定位实际上就是通过四颗已知位置的卫星来确定 GPS 接收器的位置。

如图 7-1 所示，图中的 GPS 接收器为当前要确定位置的设备，卫星 1 至卫星 4 为本次定位要用到的四颗卫星。

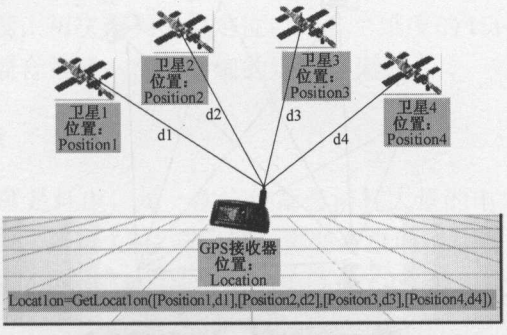


图 7-1　GPS 的接收原理

GetLocation()的功能是由硬件芯片来完成的，返回的位置信息为：Position1、Position2、Position3 和 Position4。返回的距离信息为 d1、d2、d3 和 d4，即从该卫星到 GPS 接收器的距离。

之所以要用四处位置，是因为数据是存在误差的。这些误差可能导致定位精确度降低，也可能直接导致定位无效。GetLocation()函数中用四颗卫星的数据正是为了校正误差。

GetLocation()函数返回空间坐标需要转到经纬度形式的坐标，再传给应用程序使用。

2. GPS 差分定位

实际上，前面所说的只是定位原理中的一种，称为单点定位或绝对定位。如图 7-2 所示，就是通过唯一的一个 GPS 接收器来确定位置。

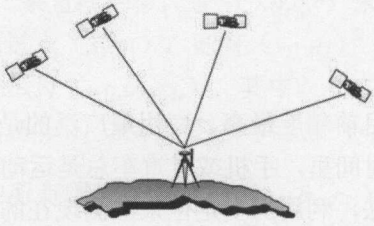


图 7-2　单点定位

目前定位精度最高的是差分定位（或称相对定位），就是通过增加一个参考 GPS 接收器来提高定位精度，如图 7-3 所示，除了 GPS 本地接收端外，还使用了一个参考的 GPS 接收器来进行差分定位。

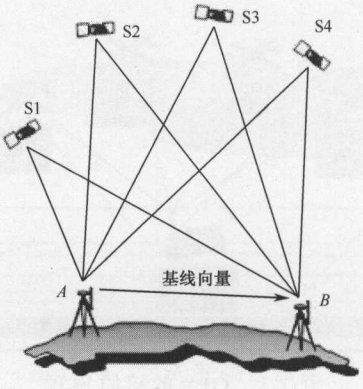


图 7-3　差分定位

3. 混合定位

混合定位就是同时使用两种以上的定位方法来进行定位。通过结合使用各种定位方法，互补短长，以达到更高的定位精度。

A-GPS 定位（辅助 GPS 定位）就是一种混合定位，是 GPS 定位技术、Wi-Fi 与 GSM 网络的结合。A-GPS 具有很高的定位精度，这是目前定位的主流技术。

GPS 定位的精度最高，可以达到 2～10m。

Wi-Fi 的定位精度可以达到 20～200m。利用 Wi-Fi 进行定位时，无须利用密码接入 Wi-Fi 网络，只需要扫一下 Wi-Fi 的标识和信号强度，就可以得到当前的位置。由于 Wi-Fi 往往可能会变换位置，所以定位服务的提供商（高德地图或者百度地图）往往需要每天对 Wi-Fi 的位置信息进行更新，比如，每天更新基站的位置信息 30%左右、Wi-Fi 位置信息 40%左右。

GSM 定位就是指移动网络定位，本质上是利用移动基站进行定位。基站定位的精度为 500～5000m，其中，中国移动和中国联通的定位精度稍好，而中国电信网络的误差最大。

在进行 A-GPS 定位时，每个 Wi-Fi 都有多个位置，对应不同的概率，同样，基站也需要由多个对应不同概率的位置。在进行定位时，首先利用聚类算法进行聚类，求最可能的位置，之后再得到目前的位置。

除聚类算法外，混合定位时也可以使用指纹算法，指纹算法在本质上是预先存储目前位置对应的各种信号的指纹信息，从而在定位时利用信号的指纹信息得到目前的位置。在得到指纹库后，利用信号进行定位时的流程可以归纳为：

计算信号的相似度，初次聚类（粗定位）→多个维度的 Hash 化后，进行指纹库的匹配→再次聚类→混合策略→得到精细的定位结果。

4. GPS 定位的缺点

GPS 定位的缺点就是耗电，每一次定位都会消耗大量的电力。由于现在的 LBS 应用主要存在于手机中，所以 LBS 应用第一需要考虑的往往就是要省电。因此，解决方案往往是在不需要 GPS 定位的地方适当地降低 GPS 的定位频率。

出于军事保密角度的考虑，目前中国地图的数据都是利用国家测绘局提供的工

具经过一定的偏移，所以，地图数据的位置和实际的位置并不相同。在这种情况下，为了定位准确，一般需要先将定位信号进行同等程度的偏移，再映射到地图上。

即使没有 GPS 信号的情况，定位也是可能的。比如：

- 汽车会利用汽车内的陀螺仪或者速度仪进行隧道内的定位预测；
- 在没有信号时的手机定位可以利用手机内的陀螺仪和当前速度（利用以往的定位时间频率和定位位置得到）进行一定的积分运算得到。

7.2 算路

从本质上说，道路计算是一个寻求最短路径的问题。最短路径分析在事故抢修、交通指挥、GPS 导航、网络游戏、曲线分段拟合等行业应用中使用非常广泛。目前大多数地图平台会把这个路径计算功能作为一个最基础的功能集成进去，如高德地图、百度地图。该功能也是 LBS 平台的基本功能之一。

最短路径的算法分为两种：遍历式算法和启发式算法。启发式算法是遍历式算法的改进，是在遍历式算法中加入启发式因子的结果，也是目前业界的主流算法。

7.2.1 遍历式算法

遍历式算法中最常用的就是 Dijkstra 算法。Dijkstra 算法是各种最短路径算法的基础。Dijkstra 算法又称为单源最短路径，所谓单源，是指在一个有向图中，从一个顶点出发，求该顶点至所有可到达顶点的最短路径问题。若要顺利实现算法，要求理解 Dijkstra 的算法，需要理解图的一些基本概念。图由结点和边构成，将结点和边看成对象，每个对象有自己的特有属性。如，在 LBS 中，一个结点必须都有 ID、横坐标、纵坐标等基本属性，边有起点结点、终点结点、长度等属性，而最短路径分析就是根据边的长度作为边的代价进行分析。

Dijkstra 的算法的评估函数为：$f(n)= g(n)$。其中，$g(n)$是对图中结点 n 的代价估计（常用的代价是从开始结点到 n 的路径长度）。

拓扑图如图 7-4 所示，其中，A 为起点，D 为终点，边上框内的数字为权值。

步骤 1：初始化图 7-4 中从 A 出发的路径集合（B、C、E）。

步骤 2：从 A 中最短路径集合（B、C、E）中找到一个最短的路径点（比如：B）

开始分析。

步骤 3：寻找从 B 出发的最短路径。

重复上述步骤，直到到达 D 点为止。

这时路径集合表中已经保存了从 A 到 D 点的最短路径集合（A、B、E、D）。

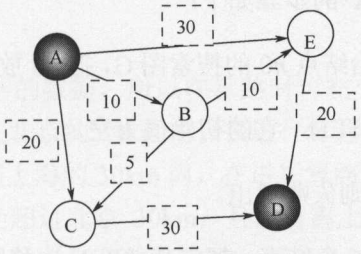


图 7-4　Dijkstra 的算法

7.2.2　启发式搜索

启发式搜索算法与广度优先的遍历式搜索类似，不同的是，它会优先顺着有启发性的结点搜索下去，这些结点可能是到达目标的最好路径。

1. 启发式搜索算法的基本思想

我们称这个过程为最优（best-first）或启发式搜索，其基本思想如下。

1）假定有一个启发式的评估函数 $f(x)$，可以帮助确定下一个要扩展的最优结点，我们采用一个约定，即 $f(x)$的值小表示找到了好的结点。这个函数基于指定问题域的信息，它是状态描述的一个实数值函数。

2）下一个要扩展的结点 n 是 $f(n)$值最小的结点（在结点扩展产生一个结点的所有后继中寻找到的）。

3）当下一个要扩展的结点是目标结点时终止过程。

最优搜索的评估函数为：$f(n)= g(n)+ h(n)$，$g(n)$是对图中结点 n 的道路代价（即从开始结点到 n 的路径长度），$h(n)$是对结点 n 的启发因子。如果 $h(n)$为 0，则 $f(n)= g(n)$，这时的搜索等同于 Dijkstra 算法。

在 LBS 的路径搜索中，$g(n)$可以设为从起点到目前点的路径长度；$h(n)$可以设为

从目前点到终点的直线长度。

2. 启发式搜索算法的步骤

当对 Dijkstra 加入不为 0 的启发式搜索因子后，这种启发式搜索算法又被称为 A*搜索算法。

如图 7-5 所示，算法 A*的步骤如下：

1）生成一个只包含开始结点 $n0$ 的搜索图 G，把 $n0$ 放在一个叫 OPEN 的列表上。

2）生成一个列表 CLOSED，它的初始值为空。

3）如果 OPEN 为空，则失败退出。

4）选择 OPEN 上的第一个结点，把它从 OPEN 中移除，并移入 CLOSED，称该结点为 n。

5）如果 n 是目标结点，顺着 G 中从 n 返回到 $n0$ 的指针找到一条路径，获得最短路径，成功退出（该指针在第 7）步建立）。

6）得到结点 n 的后继结点集 M，n 的祖先不能在 M 中。

7）对 M 的成员建立一个指向 n 的指针，把 M 的这些成员加到 OPEN 中。

8）按递增值，重排 OPEN。

9）返回第 3）步。

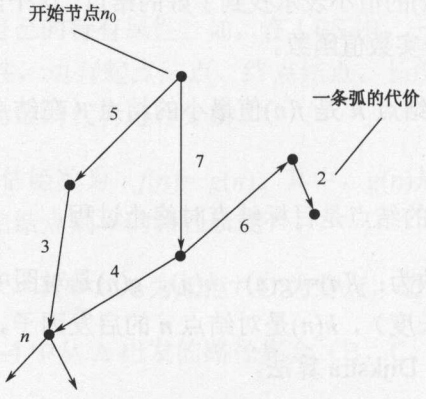


图 7-5　A*搜索算法

在实际应用中，需要注意的是，A*算法有单向 A*（从起点开始搜索，直到终点）和双向 A*（从起点、终点同时搜索，看路径什么时候会相交）之分。一般而言，对中国地图来说，双向 A* 要比单向 A* 快 15%左右。

启发式因子的设定不仅有助于地图最短路径的搜索，也有助于发现解魔方的方法等。

3. 道路搜索的加速技巧

由于道路可分为上下层的数据，所以在算路时可充分利用上下层的数据，即：如果计算从北京到上海，在北京的 30km 内，在进行算路时，既利用上层的道路，也利用下层的道路。同样，在上海的 30km 内，在进行算路时，既利用上层的道路，也利用下层的道路。但是，当超过北京 30km，且没有离上海 30km，可以只计算上层的道路。由于上层的道路少，且每条的距离长，所以计算速度可以明显加快。

为了使速度更快，可以提出一定的剪枝策略，比如：从北京到上海，如果计算的路径已经到沈阳，则明显是可以被剪枝的。

剪枝策略的具体的方法有很多，其核心是以城市为核心、以瓦片为单位来施行剪枝策略。

7.3 路径引导

路径引导（Guidance）是指在路径导航的过程，实时跟踪当前车辆所在的路径和状态，计算出当前车辆到下一引导点的距离、方向、下一道路名和目的地信息，并向自驾者发起视频和语音指令，从而不断地引导用户方便、快速地到达（POI）目的地。由于各种原因偶尔偏离规划的路径行驶，一旦系统识别车辆不再行驶在给定的路径上时，系统必须做出反应，应让车尽力回到正确的路径上，提醒自驾者的一个简单方法是在界面上显示一个方向箭头指向预定的目的地。只有车始终偏移路径行驶时，才考虑重新路径规划，让自驾者的兴趣目的地导航服务跟随无时不在。

在装有导航系统的汽车内，用户可以在导航兴趣点搜索功能画面中输入想要去的旅游路线 POI 目的地——景点，然后导航系统会自动计算出一条最佳的行车路线。在车辆行驶的过程中，系统会对前方道路上的情况加以提示。

引导点是路径引导的关键，因为在引导的过程中，最重要的就是时刻提供用户所需的路径引导信息，从而使用户能一直保持在正确的路径上。

引导点制作是制作引导点以及相关的引导信息的过程。根据实际需要，引导管理线程启动进行引导点（列表）信息制作线程执行制作任务。引导信息制作线程处理具体的引导点制作。引导信息制作的主要处理过程是在路径探索完成后进行的，通过对探索路径的遍历，产生一系列控制点或行为点（Guide Point），并准备制作具体相关的引导信息，用于显示和语音播报，从而以此为自驾者提供安全、准确的导航信息和语音指令服务。

制作过程中一个关键的概念是这些 turn-by-turn 中的“turn”控制点（或行为点），准确地说，是转弯的衍生，即引导点（Guide Point、maneuver）信息，它好比指路向导，可以产生具体的语音指令和信息。引导信息包括路口的转弯方向、下一道路名、到达距离，以及到达目的地时间和距离，当前车所在车道信息、周围的实景、路牌指示标志（signpost）等。

由于在离前方道路不同的位置、在不同等级的道路、不同类型的引导点类型，引导展现的内容和语音播报的方式都不同，这些信息主要来源于 turn-by-turn 列表。

一个实用的引导点的道路拓扑模型如图 7-6 所示。

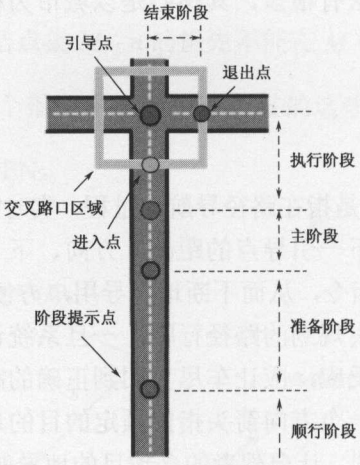


图 7-6　A*搜索算法

该引导点包括的对象如下。

- 引导点的类型（包括行为类型、方向）；
- 引导点的进、出道路线；
- 引导点的其他交叉道路（几何构成）；

- 引导阶段提示点（Phase Advice Point，简称 PaP）；
- 引导点的位置；
- 引导点的前继（对复合引导点）。

需要注意的是，当导航路线较长时，计算全部的 Maneuver（引导点对象）耗时多，并且全部的 Maneuver 占用内存数量较多，因此，在设计时使用分段计算 Maneuver 的方式。每次只生成一定距离内的 Maneuver 信息，随着车辆位置的变化，动态计算后续的 Maneuver 信息。

图 7-7 的程序中，首次计算 4 条 Maneuver，当行驶过第 1 个 Maneuver 后，更新一条 Maneuver，以此类推，直到结束。

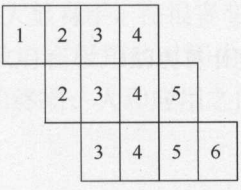


图 7-7　Maneuver 的计算

一种简单实用的 Maneuver 数据结构如下：

```
Class Maneuver{
    ArrayList*    interList;    // node 序列
    ArrayList*    drPointList;  //DRPoint 序列
    int           length;       //Maneuver 长度
    std::string   m_pstrSpeech;     //routebook
    int           icon;         //icon id
    int           m_nRouteId;   // link index
    bool      bThen;          //是否有“然后”提示
}
```

对拓扑关系较简单的一般道路而言，只需要简单的角度计算就能算出司机的行进方向。如果是环岛、广场等复杂路口，计算方法会复杂一些，这时需要考虑的角度比较多，如图 7-8 所示。

地图数据中一般存储了道路（Link）的进入、退出角度，在 Maneuver 计算时不但要使用数据提供的角度计算转向角，还需要结合路口的拓扑关系进行全面计算，才能让路口的提示更准确。如图 7-9 所示的路口情况中，带箭头的粗线是导航路线，无箭头的线是路口的拓扑路线，为了使导航提示音更准确，计算转向提示需要综合

考虑 α、β、γ 三个角度。

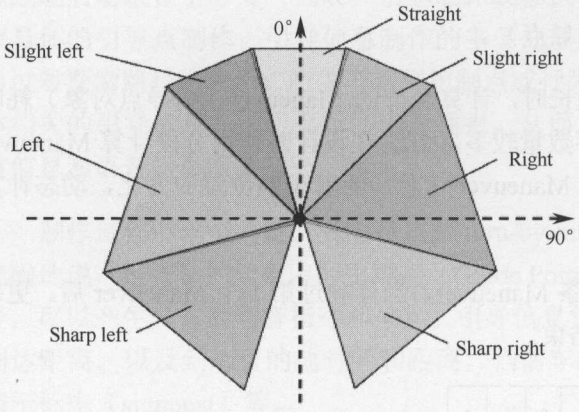


图 7-8　导航的角度计算方法

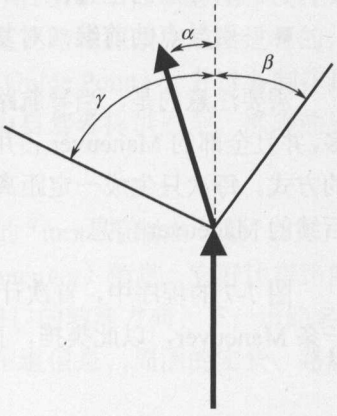


图 7-9　路口角度

7.4　TMC

TMC（Traffic Message Channel）即实时交通信息，也就是我们通常意义上所说的道路畅通或拥堵与否。

如图 7-10 所示的粗线部分就是 TMC 信息。

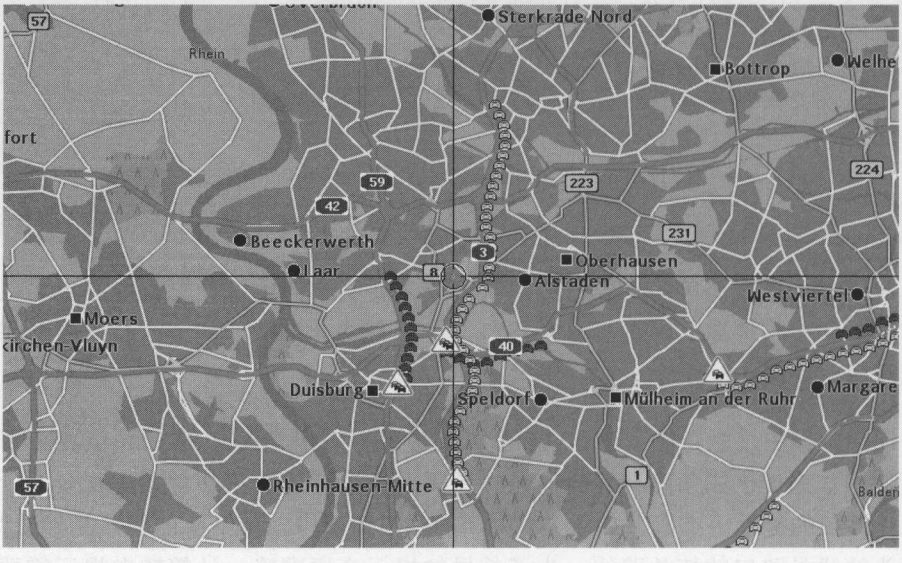


图 7-10　TMC 信息

1. TMC 信息的来源

（1）通过带 GPS 功能的手机和其他移动设备

通过带 GPS 功能的手机及其他移动设备收集位置信息和移动速度信息，根据收集到的大量数据得到某个路段的交通情况。Google、高德地图等 TMC 信息服务的提供商主要采取这种方式来得到 TMC 信息。以 Google 为例，Google 收集数据的方式是匿名的，这与你是否打开 GoogleMap 获取地址有关。另外，Google 还有预测交通路况的功能。

该方法需要大量的终端设备做支撑才能获得更加准确的数据，最近，Google 收购 Waze 的举措就是对此项功能的增强，Google 的第四大收购行动在飓风桑迪袭击美国时，社会化导航产品 Waze 大显身手，在很多交通路网瘫痪的情况下，通过用户实时分享各种路况数据，美国政府在极短的时间内就完成了各种救援力量与物资的调度，充分体现了大数据时代的终端、人和应用之间群体协同的巨大力量。

（2）通过传感器

在马路上铺设线圈，车辆碾压通过线圈时，线圈产生感应电流，通过前后线圈感应电流产生时间的不同来测定当前车速，将这些数据上传至政府交通信息的服务器，高架道路电子指示牌通过读取服务器数据来实时显示交通状况。

该方法需要交通路况正常的情况下才能获得准确的数据，如果遇到特别恶劣的天气，或者出现交通事故的情况，就无法捕获到正确的数据，因此会做出错误的导航判断。

（3）通过路口的摄像头

通过路口的摄像头拍摄的实时图片进行智能化提取和行为分析。该方法可以作为一种辅助的测量方式，可以实时切换到事故或者意外的发生地点，确保导航判断的准确性。

（4）交通台的路况播报

交通台路况播报的数据采集原理是：路况数据主要来自浮动车辆数据的收集整理。现在的出租车、长途汽车、物流车等都装有 GPS，通过通信网络，把这些车的经纬度、车头方向、速度等信息传递到数据处理中心，就可以计算出实时路况数据。当某个网络内的车辆足够多时，这种方式得到的结果也足够精准。唯一的不足是时

效性，因为路况是随时都在变化的。按照现在的技术，数据传输延迟可控制在 3 到 5 分钟内。

（5）人工参与

若要做到更加准确的实时交通路况的预报和导航，还需要人工参与，比如，收集官方网站上的交通管制通告、122 事故报警的人员通告、当地交通广播台播报的信息、互联网的信息等。

2. TMC 模块的设计

TMC 模块主要包括以下组件。

- 交通信息，提供各种形式的消息；
- 交通信息控制（选台）；
- TMC/TPEG 原数据的获取；
- TMC/TPEG 协议解析；
- TMC/TPEG 交通消息的存储；
- 交通事件的成本计算；
- 交通信息的位置匹配；
- 地图交通信息的提供。

在设计时，通常应该从以下几点考虑。

1）考虑到路径规划一般是在一个城市范围内，从数据量上看，TMC 流量更新得并不多，事件好像是不更新的，况且有很多技术可以应用，对路径计算应该不会较大的影响。

2）交通信息的实时性，事实上，目前高德/四维的发布频率也就在 2～5 分钟之间。放在数据缓存层的数据能满足当前城市的需求即可。从内容上看，数据缓存层主要缓存的数据包括：一是道路代价（流量，事件对道路的时间延迟）；二是地图显示用的，应该是瓦片（TILE）方式组织的，考虑扩展性加入按区域进行增量更新的区域标识（update regionID）。

3）交通信息模块（进程）自身的接口，提供的交通信息（TiMessage）应该是动态更新的（有新增、删除、更新的），这样异步代价并不大；提供接口也是为了引擎灵活地控制，可以提供一个统一的不依赖于数据结构的消息列表。

4）如果数据量确实很大，由应用直接读写效率可能会有问题，则解决方法就是把数据量大的工作交给地图显示引擎完成：数据管理提供存储器（accessor）接口，TMC 进程每次请求一个矩形范围内的数据，由于地图显示进程已有此范围的数据，则 TMC 进程利用“TMC 进程与地图显示进程间的共享缓存”就可以。

5）交通信息模块中的持久性存储不同于共享缓存，主要目的是考虑到系统掉电等可以快速复位，毕竟每重新做一次交通信息的请求和收集、解析、匹配等总的代价毕竟很大，这是因为交通信息的第一次请求和收集、解析、匹配的代价通常是最大的。

6）关于交通信息在导航路径上的绑定和事件列表提供，属于 GUIDANCE（引导模块）的职责。另外，提供接口给地图和上层应用，只是引导模块不会给出详细信息。根据 ID 给出交通信息的详细数据，还需要由交通信息模块提供。

7）如果需要缓存（cache）中存储 TMC 的道路瓦片（tmcTile）快速找到对道路标识（LinkId）所对应的交通信息，可能需要做一些对应的偏移或索引处理。

3. TMC 的实现技术

TMC 一般分点、线、面的 TMC，但目前主流的 TMC 服务是线的服务，即关于道路（Link）的 TMC。

从 TMC 规格上说，一个 TMC 的位置标识 tmclocid 对应一组道路序列，每个道路可以对应多个偏移量（offset），每个偏移量区间一个交通信息（畅通或拥堵）。这一点在目前的地图数据提供商中只有一两家做到了。

但是，在 TMC 信息的实际下发中，往往是比较粗糙的，一个交通信息对应一个 TMC 的位置标识（tmclocid）。比较好的 TMC 信息会达到一条道路对应一个交通信息，即一条道路只有一个交通信息。

所以，在地图中显示 TMC 时，把 tmclocid 所代表的道路序列在地图中展开，并按照 tmclocid 所对应的交通信息赋予道路某种颜色，比如：畅通为绿色，拥堵为红色。

第 8 章

显示

LBS 应用的渲染可以分为一般的 2D 文字和图像的渲染、3D 渲染。其中，2D 文字的渲染往往是由一种显示策略决定的，比如：在团购的界面中，可能会通过网页显示的技术来显示文字；在地图中，会通过 OPENGL 来显示文字注记，在具体的文字注记的显示策略上会设定规则，从而保证视觉和应用上的合理性。

用网页来显示文字并没有什么核心技术可言。所以，本章重点描述 2D 的图像（基本显示要素，简称 BMD）和 3D 的显示。

8.1 基本显示要素

2D 图像显示的核心技术有两个：上下层分别显示和化简；三角剖分。

上下层分别显示的基本原理与 3D 中的 LOD 显示策略（见 8.2.1 节的内容）相同，在上层（比例尺较小，显示的范围较广时），为了提高渲染速度，可对上层的元素进行化简，从而提高显示渲染的速度。化简在本质上减少了多边形的顶点，所以也就减少了剖分后三角形的数量。此外，为了更清晰地说明地图上下层的设置策略，本章将详细描述一个实际的示例。

三角剖分的必要性在于：由于现在 2D 面或 3D 模型的显示一般用 OPENGL 等引擎来实现，在这些引擎中，是以三角形为单位来渲染的。所以，三角形的渲染速度高于其他多边形形状，且三角形的数量越少，渲染速度越快。在 LBS 应用中，通常将 2D 面或 3D 模型化为三角形序列后，再利用 OPENGL 渲染实现，这样会得到极高的渲染效率。

8.1.1 分层显示和渲染

我们已经知道，地图是按照比例尺分为上下层（即空间索引中四叉树的上层结点，参见 5.1.2 节的内容）的，而且按照空间索引切分后存储数据。

BMD（Base Map Display，即基本地图显示元素）是一种抽象的数据，是指用户

在地图上看到的元素。具体地说，BMD 包含以下要素。

- 高层（低分辨率）的道路（ROUTE）及形状点；
- 用于显示的文字注记；
- 海洋、湖泊、公园等背景数据（Landuse）。

显然，道路是一种特殊的地图数据，因为道路既可以用来显示，也可以用来检索。所以，在一般的交换格式的数据（即高德地图或百度地图所使用的数据）中，往往可以没有上层的道路数据，也不会把道路分为显示与引导两部分，只存储一种数据。

但是在物理格式的数据（如车载导航使用的数据：NDS、KIWI 等）中，会区分显示的道路数据（包含形状点的数据）和用于引导的道路数据（只包含道路的拓扑点、道路拓扑的角度、道路长度等引导所必需的信息）。由于用于引导的数据中并没有存储形状点，所以在计算道路时，速度增快。

BMD 显示的根本技术就是为了降低显示所需的数据量，从而提高显示时的渲染速度。按照比例尺的不同而分层的策略，就是为了实现最高的渲染速度而采用的策略。分层的策略包括以下两个层面。

- 上、下层都是空间索引的一部分，上层只不过是空间索引的上层结点，下层是空间索引的下层结点；
- 在上层（小比例尺）由于存储的区域大，所以为了快速渲染，需要利用化简来存储数据。

1. 化简策略

针对上层数据的化简，有两种常用的化简算法：DP 算法和 VSA 算法。

（1）DP 算法（利用距离）

DP 算法又称为道格拉斯算法，是一种利用距离来化简的算法，其主要步骤如下。

1）在曲线首尾两点 A、D 之间连接一条直线 AD，该直线为曲线的弦。

2）得到曲线上离该直线段距离最大的点 B，计算其与 AB 的距离 d。

3）比较该距离与预先给定的阈值 threshold 的大小，如果小于 threshold，则该直线段作为曲线的近似，该段曲线处理完毕。

4）如果距离大于阈值，则用 B 将曲线分为两段 AB 和 BD，并分别对两段曲线进行步骤 1）至步骤 3）的处理。

5）当所有的曲线都处理完毕时，依次连接各个分割点形成的折线，即可作为曲线的近似。

（2）VSA 算法（利用面积）

VSA 算法是利用面积来化简的一种算法，其主要步骤如下。

1）依次连接 Link 上的所有点，得到其组成的三角形的面积。

2）比较该面积与预先给定的阈值 threshold 的大小，如果小于 threshold，则此中间点需要去除，该段曲线处理完毕。

3）如果面积大于阈值，则此中间点保留。

4）当所有的曲线都处理完毕时，依次连接各个分割点形成的折线，即可作为曲线的近似。

上述两种算法都是常见的对形状点进行化简的方法，其中一种是利用对给定直线的偏移，另一种是利用面积。这两种算法在本质上都是一种拟合算法，只不过是一种简易形式的拟合。

2. 分层策略

为了说明 BMD 在上下层的处理策略，下面举一个实际的例子。

问题的引出：在多比例尺地图中，几何线形状要素的显示层级设置越完善，则显示出来的细节越丰富，能提供的地图信息也越优，同时地图的显示效果也会更好。

而这种输出数据的显示层级实现与输入数据的显示层级设置往往有很大关系，在输入数据和输出数据的显示层级截然不同时，固定的配置文件或人工修改输入数据都不能解决显示所出现的缺陷问题。

目的：需要多比例尺的电子地图，并使线元素的显示层级实现最优对应。

生成方法包括以下步骤。

步骤 1：过滤原始数据中的线元素（比如道路等），对某些特定元素配置其固定的显示层级（比如国境线）。

- 对于道路 Link，由于这通常是由 FunctionRoadClass 来决定其显示层级的，所以其显示层级是固定的，这类线元素需要过滤掉；
- 对国境线进行特殊处理，将其显示层级设定为全层级显示；
- 对于省界线和其他 Border Line，依据项目需要进行层级设定；
- 对于其他种类的线，存储为一个专门的种类文件，以便进行后续流程的处理操作。

步骤 2：对过滤后的原始地图数据中的几何线形状，利用 B 树进行索引处理。

在原始的多比例尺地图数据中，每个线元素都是由多个形状点组成的，比如，对线元素 L1，其由形状点序列（p1，p2，p3，p4）组成，如图 8-1 所示。

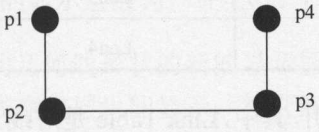


图 8-1　线元素

之后根据线元素的构造形成以下表结构。

Link Table 结构如表 8-1 所示。

表 8-1

ID（主键）	LID（线元素 ID）	SEQ（形状点在线元素中的序列号）	PID（形状点 ID）

Link Table 记录了线元素与组成它的形状点之间的构成关系。

PointTable 表结构如表 8-2 所示。

表 8-2

ID（主键）	PID（形状点 ID）	Lon（经度坐标）	Lat（纬度坐标）

Point Table 记录了点与坐标之间的构成关系。

之后基于初始化的线元素坐标信息，填充 Link Table、Point Table 的内容，例如，根据图 8-1 所示的线元素填充 Link Table、Point Table 的内容如表 8-3、表 8-4 所示。

表 8-3

ID（主键）	LID（线元素 ID）	SEQ（形状点在线元素中的序列号）	PID（形状点 ID）
1	L1	1	p1
2	L1	2	p2
3	L1	3	p3
4	L1	4	p4

表 8-4

ID（主键）	PID（形状点 ID）	Lon（经度坐标）	Lat（纬度坐标）
1	p1	Lon1	Lat1
2	p2	Lon2	Lat2
3	p3	Lon3	Lat3
4	p4	Lon4	Lat4

并对以上表建立 B 树索引，其中，Link Table 是对线元素 ID 建立索引，Point Table 对形状点的 ID（PID）建立索引。

经过上面的处理后，线元素已经被分解为多个可索引的形状点。

步骤 3：对每个线元素的长度进行计算，并设定长度与对应层级的对应关系。

在该步骤中，输入为某一特定线的形状点，对本线元素计算长度。

实现计算长度的具体过程如下。

步骤 3a：将线元素的形状点依次排列。

步骤 3b：计算每段相邻的形状点的欧几里得长度，得到三段长度：L_{p1p2}、L_{p2p3}、L_{p3p4}。

步骤 3c：将形状点的长度累加，线元素的长度 $L_{L1}=L_{p1p2}+L_{p2p3}+L_{p3p4}$。

之后将长度记录为：$L_{L1}=720$（米）。

设定长度与显示层级的对应关系，如果输出数据设为 10 层，则对应关系如表 8-5 所示。

表 8-5

层级	Lv10	Lv9	Lv8	Lv7	Lv6	Lv5	Lv4	Lv3	Lv2	Lv1
长度	0	150	300	600	1500	2400	3600	5400	10000	15000

其中：

- 长度的单位统一为米（m）；
- Lv10 对应的是最小比例尺；
- Lv1 对应的是最大比例尺。

步骤 4：根据所记录的对应关系，利用长度设定线元素所对应的层级。

由于 L1 的长度为 720 米，通过显示层级的对应关系表可以得出：L1 所对应的显示层级为 7 层。同时，在电子地图领域，线处理的一般原则是在高层显示的线在低层也显示，则 L1 线所对应的显示层级为：在 Lv7、Lv8、Lv9、Lv10 层显示。

通过上述步骤构造出来的几何线形状的新的显示层级符合较长的线可以在上层显示的原则。而且使较长的线在底层也能显示，符合线元素处理的一般原则。在上述技术方案中，通过调整长度与显示层级的对应关系的设定值，可以基于原始的多比例尺地图数据，提高生成多比例尺的线元素电子地图的质量。

当地图的显示层级设置好后，只要客户端在渲染地图元素时调用正确的显示层级，则地图的显示元素就不会有压盖问题，显示的效果如图 8-2 所示。

图 8-2　地图的显示元素

8.1.2 三角剖分

虽然曲线、曲面等有精确的方程来表示，但是在计算机中，只能用离散的方式来逼近。如曲线可用直线段来逼近，而曲面可用多边形或三角形来表示。用多边形网格表示曲面是设计中经常使用的形式，可以根据应用要求选择网格的密度。由于三角形是平面域中的单纯图形，与其他平面图形相比，其有描述方便、计算机处理简单等特性，很适合对复杂区域进行简化处理。利用三角形面片表示的曲面在计算机图形学中也称为三角形网格。

三角剖分就是一种将复杂图形变为三角形序列的方法。三角剖分在数字图像处理、计算机三维曲面造型、有限元计算、逆向工程等领域有着广泛的应用，在 LBS 领域一般用于增加渲染速度，方便点定位。

Voronoi 图和 Delaunay 三角剖分是常见的两种空间切分的方法，其应用领域十分广泛。

- 几何建模——用来寻找三维曲面“好的”三角剖分；
- 有限元分析——用来生成“好的”有限元网格；
- 地理信息系统——用来进行空间领域分析；
- 结晶学——用来确定合金的结构；
- 人类学和考古学——用来确定氏族部落、首领权威、居住中心或堡垒等的影响范围；
- 天文学——用来确定恒星和星系的分布；
- 生物学生态学和林学——用来确定动植物的竞争；
- 动物学——用来分析动物的领地；
- 机器人学——用来进行运动轨迹规划（在有障碍物的情况下）；
- 市场学——用来建立城市的市场辐射范围。

1. Voronoi 图

Voronoi 于 1908 年将平面点的邻域问题扩展到高维空间。半空间定义 Voronoi 图：给定平面上的 n 个点集 S，$S=\{p_1, p_2, \cdots, p_n\}$，定义如下：

$$V(p_i) = \underset{i \neq f}{I} H(p_i, p_j)$$

p_ip_j 连线的垂直平分面将空间分为两半，$V(p_i)$表示比其他点更接近 p_i 点的轨迹是

n-1 个半平面的交，它是一个不多于 n-1 条边的凸多边形域，称为关联于 p_i 的 Voronoi 多边形或关联于 p_i 的 Voronoi 域。如图 8-3 所示为关联于 p_1 的 Voronoi 多边形，它是一个四边形，而 n=6。

一个 Voronoi 多边形的例子如图 8-3 所示。

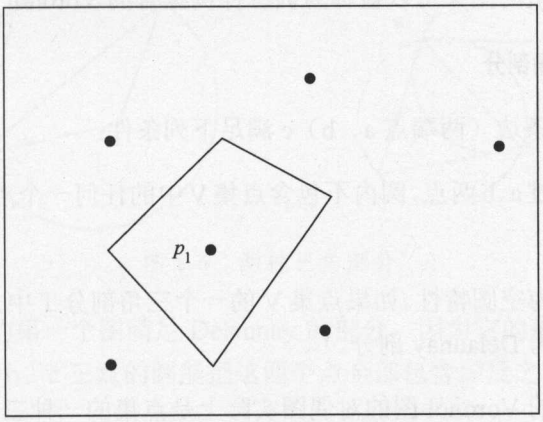


图 8-3　Voronoi 多边形

对于点集 S 中的每个点，都可以做一个 Voronoi 多边形，这样的 n 个 Voronoi 多边形组成的图称为 Voronoi 图，记为 Vor(S)，如图 8-4 所示。

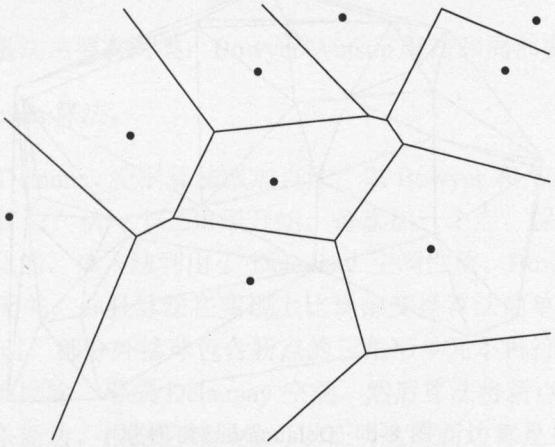


图 8-4　Voronoi 图

图 8-4 中的顶点和边分别称为 Voronoi 顶点和 Voronoi 边。显然，Vor(S)划分平面成 n 个多边形域，每个多边形域 $V(P_i)$包含 S 中的一个点，而且只包含 S 中的一个

点，Vor(*S*)的边是 *S* 中某点对垂直平分线上的一条线段或半直线，从而为该点对所在的两个多边形域所共有。Vor(*S*)中有的多边形域是无界的。

我们回想在第 6 章中 *K*-近邻算法的知识，实际上，Voronoi 图就等于 1 近邻算法在整个实例空间上所生成的决策面。这里的决策面是指围绕每个训练样例的凸多边形的合并。这种类型的图又可以被称为训练样例集合的 Voronoi 图。

2. Delaunay 三角剖分

假设 E 中的一条边（两端点 a、b）e 满足下列条件：

存在一个圆经过 a、b 两点，圆内不包含点集 V 中的任何一个点，则 e 称为 Delaunay 边。

这一特性又称为空圆特性。如果点集 V 的一个三角剖分 T 中只包含 Delaunay 边，那么该三角剖分称为 Delaunay 剖分。

最近点意义下的 Voronoi 图的对偶图实际上是点集的一种三角剖分，该三角剖分就是 Delaunay 剖分（表示为 DT(*S*)），其中，每个三角形的外接圆不包含点集中的其他任何点。因此，在构造点集的 Voronoi 图之后，再做其对偶图（即对每条 Voronoi 边）做通过点集中某两点的垂直平分线得到 Delaunay 三角剖分，如图 8-5 所示。

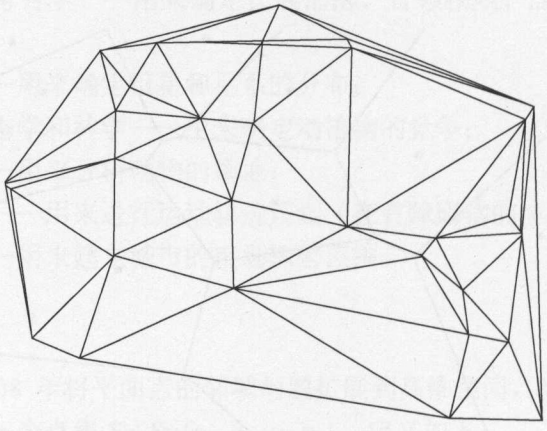


图 8-5　Delaunay 三角剖分

（1）Delaunay 三角剖分的特性

在点集的所有三角剖分中，Delaunay 三角剖分使得生成的三角形的最小角达到

最大（max-min angle）。因为这一特性，对于给定点集的 Delaunay 三角剖分总是尽可能避免“瘦长”三角形自动向等边三角形逼近。

当有四个点时，可能会有两种三角剖分，如图 8-6 所示。

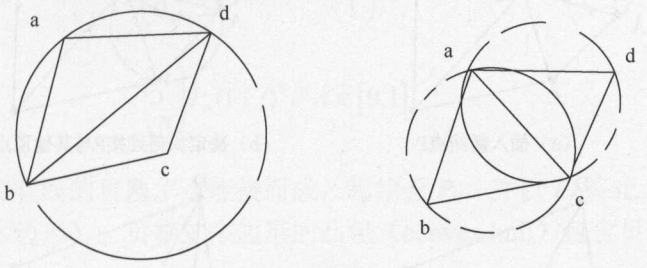


图 8-6　两种三角剖分

这两种剖分中的第一个图满足 Delaunay 的剖分，因为它的最小角最大，而且在这种剖分下，过 a、b、c 三点的圆能把这四个点全部包含，反之则做不到。

（2）Delaunay 算法的实现方法

目前 Delaunay 算法的实现方法分为：有扫描线法（Sweepline）、随机增量法（Incremental）和分治法（Divide and Conquer）等。其中，随机增量法是最主要的 Delaunay 三角剖分方法。

经典的随机增量法主要有两类：Bowyer/Watson 算法和局部变换法。

1）Bowyer/Watson 算法。

该算法又称为 Delaunay 空洞算法或加点法，以 Bowyer 和 Watson 算法为代表，是目前应用最多的算法。从一个三角形开始，每次加一个点，保证每一步得到的当前三角形是局部优化的。该方法利用了 Delaunay 空洞性质。Bowyer/Watson 算法的优点与空间的维数无关，并且算法在实现上比局部变换算法简单。该算法在新点加入到 Delaunay 网格时，部分外接球包含新点的三角形单元不再符合 Delaunay 属性，则这些三角形单元被删除，形成 Delaunay 空洞，然后算法将新点与组成空洞的每一个顶点相连生成一条新边，根据空球属性可以证明这些新边都是局部 Delaunay 的。因此，新生成的三角网格仍是 Delaunay 的。

Bowyer/Watson 算法的流程如图 8-7 所示。

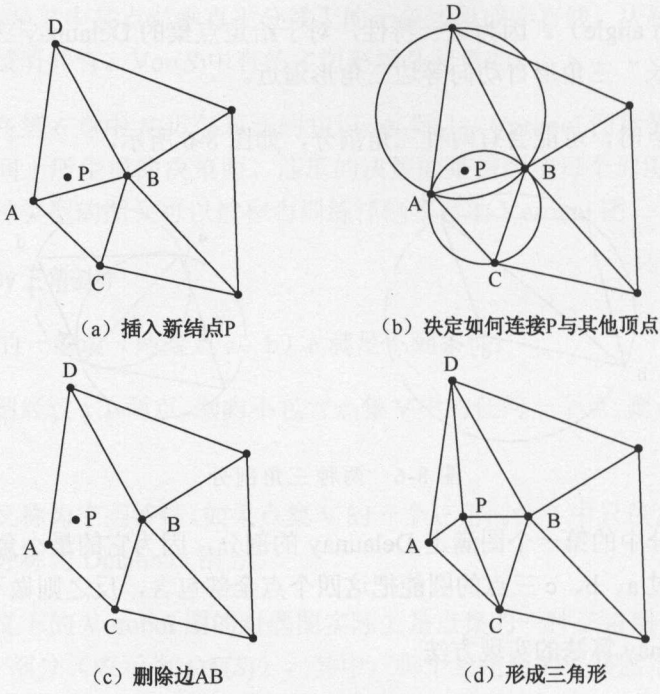


图 8-7　Bowyer/Watson 算法

2）局部变换法。

局部变换法又称为换边法或换面法。当利用局部变换法实现增量式点集的 Delaunay 三角剖分时，首先定位新加入点所在的三角形，然后在网格中加入三个新的连接该三角形顶点与新顶点的边，若该新点位于某条边上，则该边被删除，四条连接该新点的边被加入。最后，通过换边方法对该新点的局部区域内的边进行检测和变换，重新维护网格的 Delaunay 性质。局部变换法的另一个优点是其可以对已存在的三角网格进行优化，使其变换成为 Delaunay 三角网格，该方法的缺点则是当算法扩展到高维空间时变得较为复杂。

8.1.3　曲线拟合

曲线拟合可以分为：贝塞尔拟合、螺线拟合等。主要是因为表达的参数曲线不同。曲面拟合与曲线拟合类似，只是参数方程不同，往往用最小二乘法来处理。所以，只要明白了曲线拟合，就明白了曲线、曲面拟合的本质。

在曲线拟合中，最主要的一类拟合为贝塞尔拟合。

1. 贝塞尔拟合

η 阶贝塞尔曲线可做如下推断，给定控制点 P_0、P_1、…、P_n，其贝塞尔曲线为：

$$B(t)=\sum_{i=0}^{n}\binom{n}{i}P_i(1-t)^{n-i}t^i=\binom{n}{0}P_0(1-t)^n t^0+\binom{n}{1}P_1(1-t)^{n-1}t^1+\cdots\binom{n}{n-1}P_{n-1}(1-t)^1 t^{n-1}+$$

$$\binom{n}{n}P_n(1-t)^0 t^n, t\in[0,1]$$

多边形以带有线的贝塞尔点连接而成，起始于 P_0，并以 P_n 终止，称为贝塞尔多边形（或控制多边形）。贝塞尔多边形的凸包（convex hull）包含贝塞尔曲线。

给定点 P_0、P_1，线性贝塞尔曲线只是两点之间的一条直线，这条线由以下公式给出：

$$\mathrm{B}(t)=P_0+(P_1-P_0)t=(1-t)P_0+P_1, t\in[0,1]$$

且其等同于线性插值。

二次方贝塞尔曲线的路径由给定点 P_0、P_1、P_2 的函数 B（t）决定：

$$\mathrm{B}(t)=(1-t)^2P_0+2t(1-t)P_1+t^2P_2, t\in[0,1]。$$

TrueType 字型就运用了以贝塞尔样条组成的二次贝塞尔曲线。

二次贝塞尔曲线如图 8-8 所示。

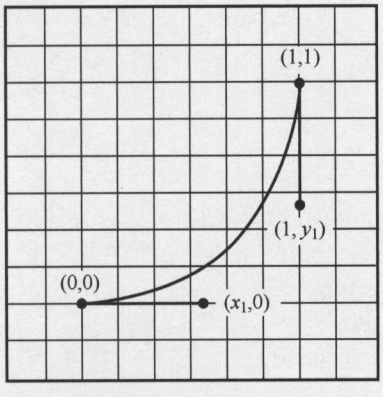


图 8-8　二次贝塞尔曲线

P_0、P_1、P_2、P_3四个点在平面或在三维空间中定义了三次方贝塞尔曲线。曲线起始于P_0，走向P_1，并从P_2的方向来到P_3。

曲线的参数形式为：

$$\mathrm{B}(t)=P_0(1-t)^3+3P_1t(1-t)^2+3P_2t^2(1-t)+P_3t^3,t\in[0,1]$$

目前的画图系统（如PS、Asymptote和Metafont）都运用了以贝塞尔样条组成的三次贝塞尔曲线，用来描绘曲线轮廓。

三次贝塞尔曲线非常适合用来绘制光滑连续的曲线，因为只需要非常稀疏的数据集就能完整地绘制那些需要精确控制的曲线。有些看上去很简单的曲线（例如圆）是无法用贝塞尔曲线或分段贝塞尔曲线精确描述的。可以用四段三次贝塞尔曲线模拟圆，每一段是一个四分之一圆。更一般的情况是可以用 n 段三次贝塞尔曲线模拟圆。

产生多段三次贝塞尔曲线的代码如下：

```
typedef struct
{
   float x;
   float y;
}
Point2D;

/*
 cp[0], P0
 cp[1], P1
 cp[2], P2
 cp[3], P3
 0 <= t <= 1
*/

Point2D PointOnCubicBezier( Point2D* cp, float t )
{
   float   ax, bx, cx;
   float   ay, by, cy;
   float   tSquared, tCubed;
   Point2D result;
```

```
    cx = 3.0 * (cp[1].x - cp[0].x);
    bx = 3.0 * (cp[2].x - cp[1].x) - cx;
    ax = cp[3].x - cp[0].x - cx - bx;

    cy = 3.0 * (cp[1].y - cp[0].y);
    by = 3.0 * (cp[2].y - cp[1].y) - cy;
    ay = cp[3].y - cp[0].y - cy - by;


    tSquared = t * t;
    tCubed = tSquared * t;

    result.x = (ax * tCubed) + (bx * tSquared) + (cx * t) + cp[0].x;
    result.y = (ay * tCubed) + (by * tSquared) + (cy * t) + cp[0].y;

    return result;
}


void ComputeBezier( Point2D* cp, int numberOfPoints, Point2D* curve )
{
    float  dt;
    int    i;

    dt = 1.0 / ( numberOfPoints - 1 );

    for( i = 0; i < numberOfPoints; i++)
       curve[i] = PointOnCubicBezier( cp, i*dt );
}
```

2. 螺线拟合

螺线（spiral）拟合虽然没有贝塞尔拟合那么重要，但也是很重要的一类拟合。螺线拟合在本质上属于一种特殊的曲线拟合，由于高速道路的转弯往往是利用螺线型的，所以螺线拟合在道路拟合上应用广泛。螺线多用于缓和曲线，以及缓和直路线与圆曲路线之间曲线变化的作用。比如，一段道路可能是图 8-9 所示的形式。

羊角螺线（clothoid）是最常用的一类螺线，其大致形式如图 8-10 所示。

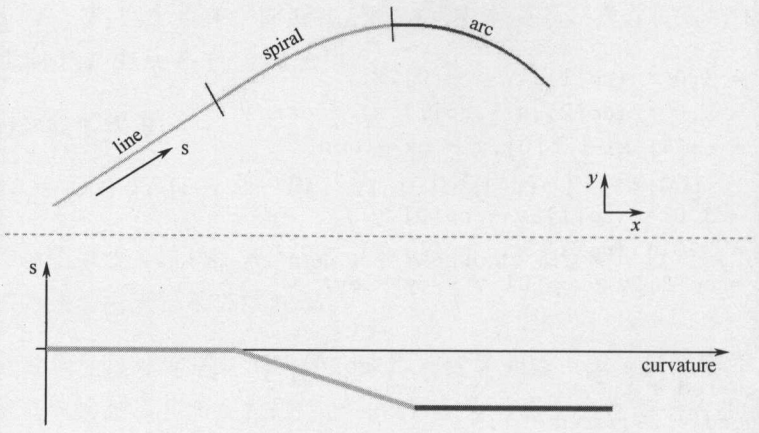


图 8-9　螺线

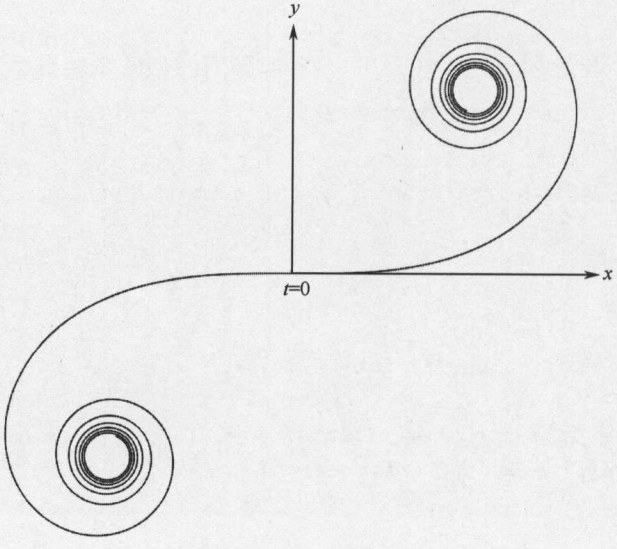


图 8-10　羊角螺线

数学形式为：

$$S(x)=\int_0^x \sin(t^2)\mathrm{d}t=\sum_{n=0}^{\infty}(-1)^n\frac{x^{4n+3}}{(4n+3)(2n+1)!}$$

$$C(x)=\int_0^x \cos(t^2)\mathrm{d}t=\sum_{n=0}^{\infty}(-1)^n\frac{x^{4n+1}}{(4n+1)(2n)!}$$

其中，$C(x)$、$S(x)$为 Fresnel 积分。参数方程的参数 t 也是螺线与该点的曲率：$k(t)=t$。如图 8-11 所示。

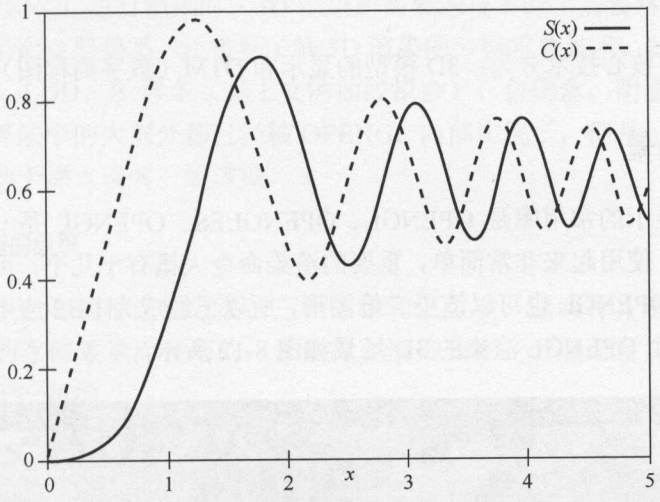


图 8-11　羊角螺线方程

由于螺线上每一处的曲率都不相同，所以，只要知道了一个螺线的首尾曲率，这个螺线的形状就可以确定。

螺线拟合或贝塞尔拟合的方法均类似，都属于曲线拟合的范畴，都可以用最小二乘法进行拟合。在拟合的时候，为了保证一阶或者二阶连续，会设定一些限制条件，具体如下。

- 进行贝塞尔拟合时，由于可以采用首尾作为控制点，在具体的拟合中，可以采取分段拟合或者全局拟合的方法；
- 对于螺线的拟合，如果需要全局拟合，可以先求首尾的曲率，之后再利用三种曲线（圆弧、直线、螺线）进行拟合，如果误差太大，而无法拟合，则将原曲线分为两段曲线来拟合。在进行分段拟合时，得到多个分段曲线后，需要对直线、圆、螺线进行拼接，为了保证拼接的圆滑，可以对首尾的曲率进行调整，从而保证一阶或者二阶连续。

需要注意的是，由于在分段拟合时，每段曲线往往都可以是直线、圆、螺线的一种或者多种，则为了在拼接这些分段曲线时得到最优的全局曲线，可以利用算路（见 7.2 节）的方法对直线、圆、螺线按照其曲率赋予不同的代价，从而求得最优的

各个分段拼接曲线。

8.2 3D 显示

3D 显示的核心技术分为：3D 模型的显示和 DTM（数字高程图）的显示。

8.2.1 3D 场景

3D 场景显示的常用库是 OPENGL、OPENGLES。OPENGL 是一个已经封装好的 3D 渲染库，使用起来非常简单，重要的渲染命令大概有十几个，可以认为是一个状态机。由于 OPENGL 也可以渲染二维图形，所以已经成为 LBS 应用在渲染时最重要的工具，一个 OPENGL 渲染的 3D 场景如图 8-12 所示。

图 8-12　3D 场景

在 LBS 发展的早期，由于当时的手机或电脑里还没有 OPENGL 的相关库，所以，当需要显示地图数据时，有以下两种技术。

- 利用 Windows 等本身的渲染库来对地图本身的点、线、面进行渲染。这种渲染的效果并不理想，渲染的速度较慢；
- 先利用服务器将地图数据渲染成图片，之后给客户端传输图片进行渲染。这种渲染本质上是对图片进行显示，所以速度够快。但是，由于图片的容量较大，所以依赖于传输的网络速度。

在 LBS 发展的当前阶段，由于 OPENGL 已经成为手机的主流配置，所以，一般

利用 OPENGL 对矢量数据进行渲染。渲染的速度也够快，而且网络数据的传输量也很小。

在利用 OPENGL 进行渲染时，3D 渲染所需要的技术除了需要对 OPENNGL 的十几个常用渲染命令要熟悉，还需要了解 3D 渲染的一些重要技术，如：三维模型的组织、BSP 树、LOD、B 样条（见上文的曲线拟合）、包围盒、阴影算法、光照效果。尽管这些算法中的大部分都已经被 OPENGL 内部实现了，但是了解这些知识的内部原理，有助于深入理解三维渲染。

1. 三维模型的组织

三维模型一般有两种组织方法：三角条带和三角扇。这两种方法都是为了降低三维模型顶点的存储数量而使用的，都能提高渲染效率。一般而言，三角条带的渲染效率要稍高于三角扇。

2. BSP 树

BSP 树是一种空间二分树，是为了进行视线范围（视见体）的背面筛选而准备的。BSP 树在本质上是一种三维空间的 K-d 树，对物体或物体上的所有面进行 K-d 树组织。在渲染时，通过视见坐标系对 K-d 树进行筛选，从而得到离视点最近的物体，或没有被遮挡的面。

出现在视线范围的背面的多边形或三角形由于没有出现在 BSP 树中，所以将不被渲染，从而提高了渲染的速度。如果用 OPENGL，就不需要这种背面筛选的方法，因为 OPENGL 内部实现的 Z 缓冲算法（把距离视见体不是最近的面删除）与这种 BSP 的做法类似。

3. LOD

LOD（level of display）也称为层次细节模型，是一种实时三维计算机图形技术，最先由 Clark 于 1976 年提出，其工作原理如下。

视点离物体近时，能观察到的模型细节丰富；视点远离模型时，能观察到的细节逐渐模糊。所以，在存储地图等海量的二维或三维的数据时，会设置多种不同分辨率的模型数据。分辨率高的模型细节丰富，数据量大；分辨率低的模型细节很少，数据量小。由于在小比例尺下，会显示很多数据，如果设置了 LOD，对应的分辨率低，对应的每个模型的数据量将较小，所以，能提高显示的渲染速度和数据的传输

速度。

LOD 示例如图 8-13 所示。

4. 包围盒

包围盒是为了碰撞检测而设定的，本质上是因为两个形状复杂的模型相互检测其是否已经发生碰撞，所需要做的计算太多。所以，就在每一个模型的周围用一个六面立方体（也可以超过六面立方体）的盒子包围起来，只要检测这两个六面立方体是否发生碰撞，就知道其是否发生了碰撞，所以降低了计算量。

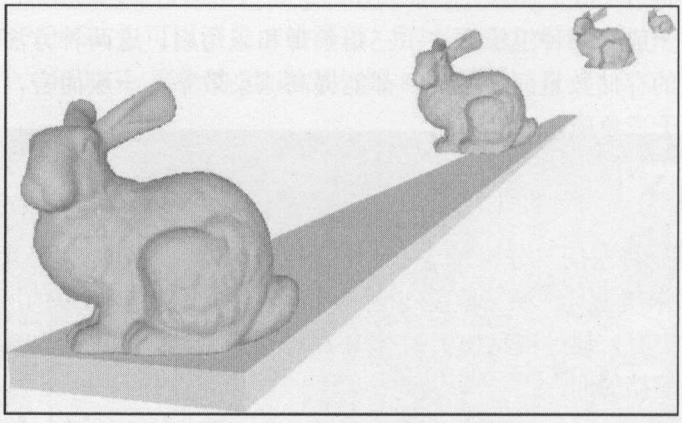


图 8-13　LOD

5. 阴影算法

阴影算法在本质上是一种矩阵的计算，即用光线和模型的轮廓边来进行矩阵计算，从而得到其阴影。模型的轮廓边可以用预处理的方法来得到，也可以在进行背面筛选（即建立 BSP 树或 Z 缓冲）时，就计算得到轮廓边。如果用 OPENGL，就不需要这种阴影算法，因为 OPENGL 内部实现的阴影算法与这种做法类似。

6. 光照效果

光照效果在本质上是通过计算面的位置和其法向量来确定的光照效果。由于三维模型的三角面是离散的，所以为了实现均匀的光照效果，需要对三角面的法向量进行插值。插值的方法有很多，最常用的是：对相邻三角面的法向量进行均匀插值。如果使用 OPENGL 库，就不需要这种光照算法了，因为 OPENGL 函数内部实现的光照算法与这种光照算法类似，在 OPENGL 函数中已经有几种光照渲染函数可以选择。

8.2.2 DTM 显示

DTM 即数字地形图，也就是通常所说的地形。LBS 应用所使用的地形数据一般是以像素的形式提供的。在渲染的时候，一般先将其三角化，比如：可以将对角的像素相连接，从而组成若干个三角形。在将像素形式的 DTM 形成三角化的模型后，DTM 的渲染实际上就等同于三维模型的渲染。一个三角化的 DTM 渲染后的效果如图 8-14 所示。

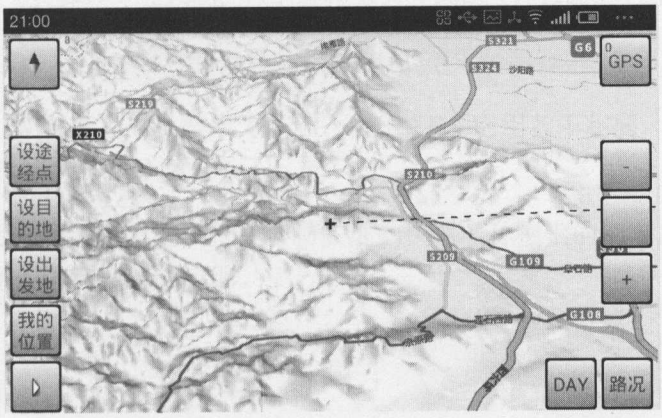


图 8-14　DTM

ORTHO（即正交影像图）也就是卫星图，可以单独显示，也可作为 DTM 的三维纹理而被显示。通常情况下，需要建立空间索引，即按照瓦片（Tile 或 Mesh）来切分卫星图。如果单独渲染 ORTHO，加载某个瓦片的 ORTHO 即可，由于已经被切分为小图片，从而能保证在渲染的时候只加载需要的卫星图，提高渲染的速度。

地形中也有 LOD 算法。DTM 的 LOD 算法可以分为：非连续 LOD 模型、连续 LOD 模型和结点 LOD 模型。

1. 非连续 LOD 模型

非连续 LOD 模型是目前应用最多的地形或 3D 的 LOD 模型。它实质上保存了原始模型的多个 LOD 副本，每个副本对应某一特定的分辨率，所有的副本构成一个金字塔模型。该模型的优点是不必在线生成模型，可视速度快；缺点是数据冗余大，容易引起几何数据的不一致性，而且由于不同的分辨率之间没有任何关联，不同分辨率间的转换易引起视觉上的间跳现象。

2. 连续 LOD 模型

它是在某一时间只保留某一分辨率的模型，在实际应用中根据需要，利用计算机采用一定的算法实时生成另一分辨率的模型。该模型的优缺点正好与不连续 LOD 模型相反，即优点是没有数据冗余，能够保证几何数据的一致性和视觉连续性；缺点是需要在线生成不同分辨率的模型，算法设计较复杂，可视速度慢。

第 9 章

搜索

搜索是用户和 LBS 交互的主要渠道。各种 LBS 应用有所不同，但是搜索技术大致可归纳为以 POI（兴趣点）为基础，以名称搜索为表现形式，以推荐系统为核心技术。

兴趣点是 LBS 应用中用户感兴趣的点的统称。

推荐系统以相似度和分类技术为基础。由于 LBS 应用各不相同，所以推荐系统在实际的应用中会有所不同。

为了使读者能更深刻地明白搭建容易实现的快速名称搜索的细节，9.3 节将详细描述两种有创意的快速搜索的实现。

9.1 兴趣点

POI（兴趣点）是指用户所感兴趣的点（商铺、地址等），为了实现 POI 的快速检索，可以对数据进行线性排序，也可以建立空间索引。

POI 的名称由于可以重复，所以为了减少存储的容量，提高检索的速度，可以对所有的名称进行统一存储。

与名称相同，由于 POI 的图标也可以重复，所以为了减少存储的容量，提高检索的速度，可以对所有的图标进行统一存储。一幅地图数据存储不到 100 个图标就满了，而 POI 只需要有一个对图标的引用即可。

在 LBS 的应用中，POI 的深度信息是最重要的信息。深度信息如：店铺的评价、店铺的营业时间，或者地图中需要更新的数据（如从网络得到的 POI 数据）等。可以想象，未来的地图数据必然是互动的地图，也必然是联网的地图。所以，未来的地图数据中有可能很大一部分是互动的深度数据，如从其他网站（淘宝、天猫）得到的深度信息。

9.2 推荐系统

推荐系统是指通过对用户的行为数据进行挖掘，从而向用户推荐有用的信息技术。

由于数据挖掘章节的相似度（见 6.1 节）已详细说了各种相似度的技术实现原理，下面将重点描述在设计推荐系统时要考虑的各种因素。

推荐引擎分以下两类。

第一类称为协同过滤，即基于相似用户的协同过滤推荐（尽最大可能发现用户间的相似度），以及基于相似物品的协同过滤推荐（尽最大可能发现物品间的相似度）。

第二类是基于内容分析的推荐（调查问卷、电子邮件，或者其他基于内容特征的分析）。

1. 协同过滤

基于协同过滤的推荐在本质上是计算相似度，可参见第 6 章相似度（即 6.1 节）的内容。

（1）协同过滤的种类

协同过滤可分为以下三个子类。

- 基于用户（user）的推荐：这种推荐是通过共同口味与偏好找相似邻居用户，常使用 *K*-近邻算法（见 6.2.2 节）。要达到的效果是：因为你的朋友喜欢，所以推测你可能也喜欢；
- 基于物品（item）的推荐：这种推荐是要发现物品之间的相似度，从而推荐类似的物品。要达到的效果是：因为你喜欢物品 A，又因为 C 与 A 相似，所以推测你可能也喜欢 C；
- 基于模型的推荐：这种推荐是要基于样本的用户喜好信息构造一个推荐模型，然后根据实时的用户喜好信息预测推荐。

（2）做协同过滤推荐应考虑的因素

上述几种推荐在使用时，要考虑以下因素。

- 精确度（Accuracy）：选择基于数量较少的因子来建立推荐算法；
- 效率（Efficiency）：效率；
- 稳定性（Stability）：选择基于变动频度较低的因子来建立推荐算法。例如，如果商品基本固定，用户不断变化，则选择基于 item 的算法；

- 说服力（Justifiability）：如果要考虑说服力，则优先选择基于 item 的算法，因为比较容易理解。例如，因为你喜欢三星 Galaxy，所以给你推荐 HTC One；
- 惊喜度（Serendipity）：如果要考虑惊喜度，则优先选择基于用户的推荐。

在工业界，通常采用的推荐算法如下。

- 多种算法融合+ 统一模型（Model）→划分等级（Ranking/Filtering）；
- 通过不同的输出来决定使用不同的算法。

（3）做协同过滤推荐的步骤

做协同过滤推荐时，一般要做好以下步骤。

1）若要做协同过滤，那么收集用户偏好则是关键。可以通过用户的行为诸如评分（如不同的用户对不同的作品有不同的评分，而评分接近则意味着喜好口味相近，便可判定为相似用户）、投票、转发、保存、书签、标记、评论、点击流、页面停留时间、是否购买等获得。

2）收集用户行为数据之后，接下来便要对数据进行减噪与归一化操作（得到一个用户偏好的二维矩阵，一维是用户列表，另一维是物品列表，值是用户对物品的偏好，一般是 [0,1] 或者[–1, 1]的浮点数值）。

3）计算相似用户或相似物品的相似度。

4）计算出来的这两个相似度将作为基于用户、物品的两项协同过滤的推荐。

2. 基于内容的推荐

所谓内容（Content），是指能够描述用户/物品的特征，比如：

对于物品来说，要考虑的内容特征有以下三项。

（1）电影

对于电影，需要考虑的特征如下。

- 电影：红番区；
- 类型：动作/冒险；
- 演员：成龙；
- 年份：1997。

（2）文本

对于文本，需要考虑的特征如下。

- 词性；
- 单词转化为向量（Word2vec）；
- 主题。

（3）多媒体

对于多媒体，需要考虑的特征如下。

- Audio；
- 图片像素。

总的来说，用户特征要考虑的内容如图 9-1 所示。

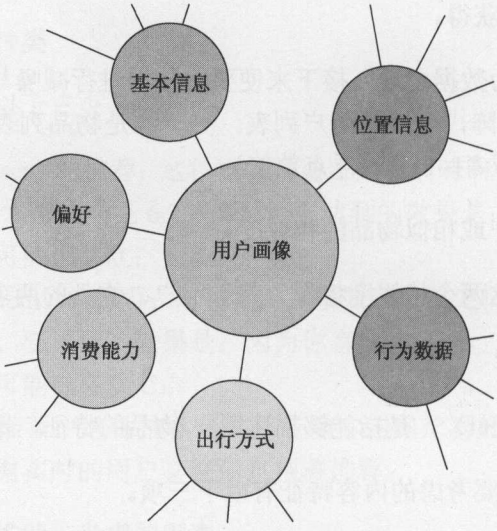


图 9-1　用户特征

将用户或物品的特征进行统一表示（例如表示成特征向量），即可利用相似度来进行等级（ranking）划分，从而推荐。

总的来说，协同过滤往往没有考虑用户或物品本身的特征，但能保证有一定的惊喜度。而基于内容的推荐能够充分应用物品（item）和用户自身的特征信息，且能够提供明确的推荐理由，但是容易推荐过于一致的结果。

9.3 名称搜索

实际上，推荐技术就是搜索技术，因为推荐技术往往是进行网页排名（谷歌发明的一种网页排名技术）的，所以推荐技术就是最核心的搜索技术。

但是，搜索技术除了出现用户最感兴趣的内容（推荐系统）外，还有效率的需求。提高搜索技术的效率往往有两种技术：大规模的哈希（Hash）技术（如 Hadoop 中所用到的海量数据存储技术）和构建树的技术。其中大规模的哈希技术在大型的搜索引擎中使用比较普遍，往往是先对文本建立分词库，然后建立一些大规模的哈希化的倒排索引等，这种技术实现起来有一定的难度。而构建树的技术相对于大规模的哈希技术而言，简单且易行。

最常见的构建树的技术就是 FTS（利用 B-树的全文搜索）或自动提示。

FTS 往往是利用与 SQLITE 的 FTS3 或 FTS4 类似的技术来建立的，本质上是一种利用 B-树的技术，使用起来比较简单。

自动提示技术大概有如下 4 种。

① 在线式技术（利用分词）：这种技术往往有一个庞大的分词数据库，利用庞大的服务器的运算能力，以单词（中文或英文）为单位生成自动提示的词汇。这种技术一般用于搜索引擎，分词的具体技术可以参见第 6 章中贝叶斯学习的分词实例。

② 离线式技术（利用分词库）：这种技术与在线式分词技术类似，一般是利用预先生成的或用户操作生成的词库。利用词库间单词的关系来自动提示下一个单词。由于这种技术所生成的词汇并非一一对应网页或文本地址。所以，这种技术往往应用于输入法。如果应用于离线式引擎，则需要将分词对应多个网页或文本。

③ 离线（不利用分词库）：这种情况下，往往使用 NVC（next valid character，下一有效字符）技术。

NVC 技术是在国外应用很普遍的一种技术，先于分词技术而产生，最早应用在车载引擎上，并一直在车载引擎上占据统治地位。这是因为 NVC 技术可以在查找到单词的同时，同步生成唯一对应的 POI 或道路的索引。所以，方法③相比于方法①、②来说，具有精确、实现简单、维护简单的特点。

进行道路或兴趣点名称的 NVC 的方法如下：

通过在数据库中预先存储所有的道路或兴趣点的名称，并预先存储 NVC 的数据内容，实现预先判断用户输入的下一个有效字符，从而可以在很短的时间内实现下一字符的自动提示。

④ 名称的全文提示。这种方法近似自动补全功能，这种自动提示方法通过用户输入的前几个字符，在名称的下拉框中提示 1 个或多个道路或兴趣点的名称字符串。

第①、②种技术所用的方法是非常好的，并且已经普遍应用在大型搜索引擎中，比如百度、谷歌或搜狗输入法。但由于这种方法的开发成本比较高，且对硬件或网络的要求也较高，所以并没有普遍应用于高端车载导航产品及低端的 LBS 产品。

没有用于低端的 LBS 产品的原因主要在于开发成本的关系。没有在高端的车载导航中成为主流是因为高端车载导航需要精确、快速更甚于界面的友好，比如，用第①、②种方法，如果不花费巨额的研发成本，往往不能满足“用户的输入词汇与对应的网页 / 文本地址一一对应”的要求。

下面举例来说，若用户要查找：望京商业楼。所寻找到符合要求的 POI 可能有多条，而用户真正希望寻找的那个 POI 因是冷门的，如果采用第①、②种做法，可能被放在第一页之后，或者被忽视。这种情况是高端汽车导航所无法接受的。

所以，目前在高端导航引擎端或低端的 LBS 应用真正使用的主流搜索技术仍是第③、④种方法。

为了帮助读者明白自动提示的具体实现，这里提供了在两个利用第③、④种方法有创意的实现，一个是按照使用频率更新的 NVC 技术，另一个是利用 NVC 技术的全文自动提示技术。明白了这两个技术后，对第①、②种方法也就掌握了其精髓。

1. 更有效的 NVC 方法

英文的下一个有效字符（NVC）只有 26 个选择，所以自动提示的下拉框并不多，尚在可承受的范围内，所以 NVC 尚有一定的实用价值。但是在中国，每个中文的下一个有效字符有可能上百个，所以用户每次都有可能面临几百个名称的选择，这会导致用户难以在每一个都看过后，再进行下一步的输入。

为了解决这一问题，可缩减下一有效字符的数量。所以，在建立 NVC 的时候，需要按照行政区划来建立。在用户输入的时候，先选择行政区划，再选择有效字符，实际的输入并不会减少，仍是所有的名称字符都输入，但是可以通过自动提示的下

一字符，在第一时间内知道自己所要寻找的道路或兴趣点在行政区划内有或者无。实际上，这种解决方案并没有真正解决下一有效字符数量太多的问题。

所以，现在对 NVC 的发展而言，如何使用户能够从下一有效字符中一眼看到常用的字符已逐步成为一个难题。

如果能提供一种基于用户行为分析的车载电子地图下一有效字符方法与装置，并通过用户对每一字符的使用频率对下一有效字符进行排序，从而实现用户最常用的有效字符排在前列，更好地满足用户的需要。

解决步骤如下。

步骤 1：设置新的 NVC 的数据结构，并构建 NVC 的数据内容，设定随用户使用而更新 NVC 的策略。

① 设置新的 NVC 的数据结构。

NVC 的数据结构随着存储方式的不同而有所不同，在连续存储 NVC 中的 node 结点时，即存储 NVC 用一块连续的内存空间时，对 NVC 树的结点的数据结构定义如下：

```
Struct  node {
Int16  character;   //中文或英文字符
Node* nextStart;   //第一个子结点的指针
Node* nextEnd;    //最后一个子结点的指针
Int  Quantity;    //使用频率
}
```

整个 node 的长度为 14 字节。

对 Node* nextStart 和 Node* nextEnd 的说明如下：

对每个非叶子结点而言，nextStart 与 nextEnd 都不为空，所指向的内容是其下一级子结点的第一个和最后一个。如果只有一个子结点，则两者的值相同。从 nextStart 到 nextEnd，总是连续存储的。比如：某结点有三个子结点，按地址从小到大排列：A、B、C。则存储在 NVC 树中时，存储的情况是：nextStart：A、nextEnd：C。在这种存储方式下，节省了存储空间，而且由于每个结点的长度都是 14 字节，所以，通过 nextStart A 的地址增加 14 字节后的地址就是下一个子结点 B 的地址。

② 构建 NVC 的数据内容。

在初始构建 NVC 时，需要将所有的名称进行排序，比如，现在需要对下面四条记录建立 NVC：

中国人；
中国菜；
中华；
解放路。

则“中国人”、“中国菜”、“中华”这三条记录有一个公用的首字“中”，所以建立为一棵 NVC 树，树的首字为“中”。由于“中国人”、“中国菜”的第二个字“国”相同，所以两者在 NVC 树的第二层公用一个汉字“国”。解放路的首字与其他三条记录都不相同，所以单独创建一棵 NVC 树。

NVC 最终构建的结果如图 9-2 所示。

在构建完 NVC 后，每个 NVC 的 node 的内容填充完毕，以非叶子结点 node1、node2 和叶子结点 node4 为例，各结点的内容如下：

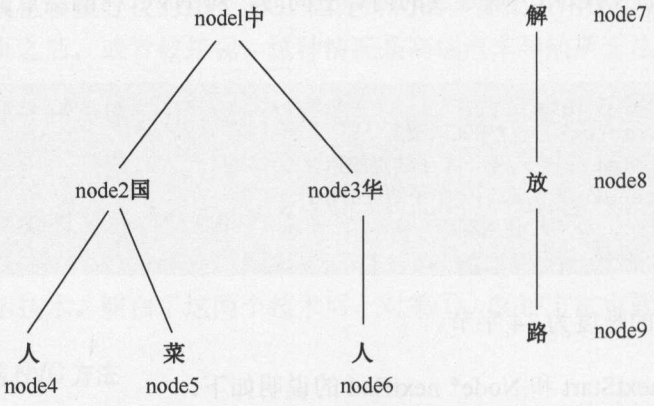


图 9-2　NVC

```
  Node 1{
Int16  中;        //中文或英文字符
Node* node2;     //第一个子结点的指针
Node* node3;     //最后一个子结点的指针
Int  0;          //使用频率
}
Node 2{
Int16  国;       //中文或英文字符
```

```
Node* node4;    //第一个子结点的指针
Node* node5;    //最后一个子结点的指针
Int  0;         //使用频率
}
Node 4{
Int16  中;      //中文或英文字符
Node* NULL;     //由于node4为叶子结点，所以没有下一级结点，第一个子结点的指针为空
Node* NULL;     //由于没有下一级结点，所以最后一个子结点的指针为空
Int  0;         //使用频率
}
```

在各结点的内容填充完毕后，进行下一步操作。

③ 设定随用户使用而更新 NVC 的策略。

在刚构建完成 NVC 时，由于每个结点的使用频率都为零，所以每个结点需要随着用户的使用而更新。具体的更新策略是指每次随着用户的输入，NVC 发生相应的变化，如：

当用户输入“中”时，则 node1 对应的频率增加 1；同理，用户输入“国”时，则 node2 的频率增加 1。

步骤 2：利用用户对每一个字符的使用频率对下一个有效字符进行排序。

对下一个有效字符进行排序的过程是一个对 NVC 进行重构的过程，在 NVC 占用连续内存存储空间的情况下，这种排序实际上在一定程度上改变了 NVC 的内部结构。

由于这种排序往往要耗费一定的时间，所以为了不影响导航引擎的快速正常运转，也可以采用周期性更新的办法。

具体的周期性更新策略如下：

设 node1、node2、node3、node4、node5、node6 的使用频率分别为 14、6、8、3、3、8。先对根结点进行更新，单独存储第二层的结点序列为序列 X1，由于原先 node1 的子结点的存储序列为：node2、node3，而在用户使用后，node2 的使用频率小于 node3 的使用频率，所以对序列 X 中 node1 的子结点排序为：node3、node2。

将序列 X 中 node1 的子结点按顺序全部写入 NVC 树的第二层。

注意：由于每一层 node 的个数是不会改变的，所以这种第二层的存储序列的改变

不会改变第三层的结点（node4、node5、node6）的存储地址，也不会破坏第三层结点的存储内容。

以此类推，对第二层的结点开始更新，从 node3 直到 node2。单独存储第三层的结点序列为序列 X2。

在对 node3 进行更新时，对 node3 在序列 X2 中的结点按照用户的使用频率进行排序，排序结束后，把序列 X 中 node3 的子结点按顺序全部写入 NVC 树的第三层，并标记其结束位置为 Y1。

在对 node2 进行更新时，node2 在序列 X2 中的结点按照用户的使用频率进行排序，排序结束后，把序列 X 中 node2 的子结点按顺序全部写入 NVC 树的第三层（紧邻 node3 的子结点，即从 Y1 处开始写）。

用此方法对 NVC 树逐层进行处理，直到 NVC 的叶子结点。

最终，按照用户的使用频率更新后的 NVC 如图 9-3 所示。

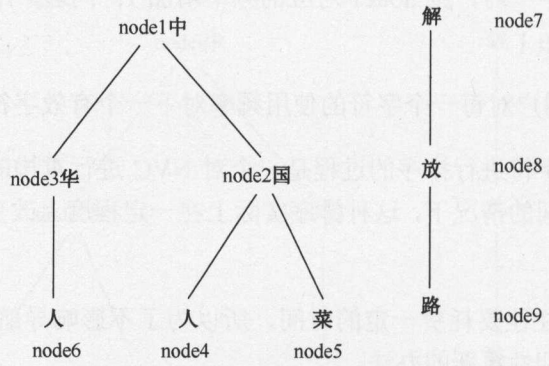


图 9-3　按照使用频率更新的 NVC

注：由于“解-放-路”这个 NVC 树中每层只有一个结果，所以不管用户的使用频率如何，其结构都不会改变。

步骤 3：在导航引擎利用名称来预先提示下一个有效字符时，通过用户已输入的字符，按照其使用频率的多少，对其自动提示的有效字符进行显示。

由于按照频率排序后的 NVC 树已经是按照用户的使用频率来更新的 NVC，所以在用户输入某一个字符时，只需要依次从 nextstart 显示到 nextend 即可。

例如，用户输入“中”时，更新前的显示如下：

第一个字符是“国”，

第二个字符是“华”。

更新后的显示如下：

第一个字符是“华”，

第二个字符是“国”。

这样，排在最前面显示的是用户输入最频繁的字符。

2. 利用 NVC 的全文自动提示技术

由于 NVC 只是提示下一个有效字符，所以用户实际上还需要在输入每个字符时都做一次选择。这种方法的效率并不高。

目前高端车载导航所使用的 NVC 方法并不是很方便，但是其他自动提示方法要么开发成本高，要么不能满足汽车导航的名称与结果一一对应的需要。

所以，我们需要一种基于用户行为分析的车载电子地图自动提示方法与装置，通过发明新型的自动提示树的数据结构，并利用用户的使用频率来对道路或兴趣点的名称进行筛选，得到用户最常用的字符串，从而更好地实现自动提示，以实现道路或兴趣点的名称补全功能。具体操作步骤如下。

步骤 1：通过设置新的自动提示树的数据结构，构建自动提示树的数据内容，设定随用户使用而更新自动提示内容的策略。

① 设置新的自动提示树的数据结构。

该数据结构随存储方式的不同而有所不同，在连续存储其结点时，即存储各结点用一块连续的内存空间时，对能自动提示的字符树的数据结构定义如下：

```
Struct  node {
Int16  character;    //中文或英文字符
List*  L;          //使用频率前几条名称记录的链表
Node* nextFirst;     //第一个子结点的指针
Node* nextEnd;     //最后一个子结点的指针
Int  Quantity;     //使用频率
}
```

整个 node 的长度是固定的，为 18 字节。

对链表 List* L 说明如下。

链表中的内容为：

```
char* Name;
```

其中，Name 对应用户使用频率。

如果是非叶子结点，则链表指向的是前某条记录。

如果是叶子结点，则链表指向的是一个字符串。

对 Int Quantity 的说明如下：

Quantity 是标志叶子结点所对应的用户使用次数。

如果是非叶子结点，Quantity 为 −1。

如果是叶子结点，Quantity 为 0 或者用户的使用次数。

对 Node* nextFirst 和 Node* nextEnd 的说明如下：

对每个叶子结点而言，nextFirst 与 nextEnd 都为空。

对每个非叶子结点而言，nextFirst 与 nextEnd 都不为空，所指向的内容是其下一级子结点的第一个和最后一个。如果只有一个子结点，则两者的值相同。从 nextFirst 到 nextEnd，总是连续存储的。比如：某结点有三个子结点，按地址从小到大排列：A、B、C。则存储在自动提示树中时，存储的情况是：nextFirst：A、nextEnd：C。在这种存储方式下，节省了存储空间，而且，由于每个结点的长度都是 18 字节，所以通过 nextFirst A 的地址增加 18 字节后的地址就是下一个子结点 B 的地址。

② 构建自动提示树的数据内容。

在初始构建自动提示时，需要将所有的名称进行排序，比如：现在需要对三条记录建立自动提示树。

阳明路；

阳明餐厅；

阳阳。

则“阳明路”、“阳明餐厅”、“阳阳”这三条记录中有一个公用的首字“阳”，所以建立为一棵自动提示树。树的首字为“阳”。由于“阳明路”、“阳明餐厅”的第二个字“明”相同，所以两者在自动提示树的第二层公用一个汉字“明”。

自动提示树是依照不同的首字符而建立不同的自动提示树的。也就是说，首字符相同的名称建立为一条自动提示树。

自动提示树最终构建的结果如图 9-4 所示。

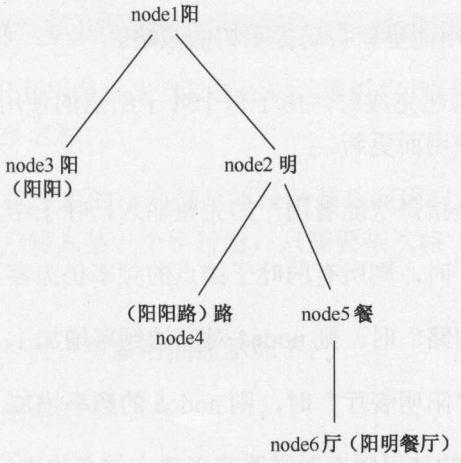


图 9-4　自动提示树

在自动提示树构建完毕后，每个自动提示树中 node 的内容填充完毕，以非叶子结点 node1、node2 和叶子结点 node4 为例，设存储前两条记录的链表，则各结点的内容如下：

```
  Node 1{
Int16  阳;   //中文或英文字符
List*  NULL;     //使用频率前几条名称记录的链表
Node* node3;   //第一个子结点的指针
Node* node2;   //最后一个子结点的指针
Int  0;         //使用频率
}
Node 2{
Int16  明;        //中文或英文字符
List*  NULL;     //使用频率前几条名称记录的链表
Node* node4;      //第一个子结点的指针
```

```
Node* node5;        //最后一个子结点的指针
Int  0;             //使用频率
}
Node 4{
Int16  路;          //中文或英文字符
List*  L4;        //使用频率前几条名称记录的链表
Node* NULL;       //由于 node4 为叶子结点，所以没有下一级结点，第一个子结点的指针为空
Node* NULL;       //由于没有下一级结点，所以最后一个子结点的指针为空
Int  0;           //使用频率
}
```

L4 指向“阳明路”。

③ 设定随用户使用而更新自动提示树的策略。

在自动提示树刚构建完成时，由于每个叶子结点的使用频率都为零，所以每个结点需要随着用户的使用而更新。

具体的更新策略是指每次随着用户的完整输入，叶子结点发生相应的变化，如：

当用户输入“阳”时，则所有的叶子结点的频率仍为零。

当用户输入“阳明路”时，则 node4 对应的频率增加 1。

同理，用户输入“阳明餐厅”时，则 node6 的频率增加 1。

步骤 2：利用用户的使用频率对道路或兴趣点的名称进行周期性筛选，得到用户最常用的字符串。具体的更新策略如下：

设叶子结点 node3、node4、node6 的使用频率分别为 5、9、8。更新的策略是从下至上，需要先对底层的第一个紧邻层的非叶子结点进行更新，这是因为树的长度为 4，由于第四层的结点为叶子结点，则其在用户输入时已经得到更新，所以先对第三层的非叶子结点进行更新。

在进行更新前，先将原来的自动提示树按照层级单独存储为 X1、X2、X3。其中，X1、X2 和 X3 分别代表第一层的结点、第二层的结点和第三层的结点。第三层的非叶子结点有：node5。

依次对第三层的结点取其频率最多的两条记录，对其结点的 List* L 进行更新。如 Node5 的前两条子结点记录为：

```
List* L = {“阳明餐厅”, 8 次}（注：只有一条记录）
```

之后，进行第二层的更新。第二层的非叶子结点为：node2。对 node2 结点的 List* L 进行更新：

```
List* L = {“阳明路，9 次；“阳明餐厅”，8 次}（注：有两条记录，则从最大至最小的顺序进行排列）
```

注意：对某层进行更新时，为了提高效率，可选下层所有结点的“List 中最大的两条记录”进行比较。

接着，进行第一层的更新。第一层的非叶子结点为：node1。对 node1 结点的 List* L 进行更新：

```
List* L = {“阳明路”，9 次，“阳明餐厅，8 次}（注：“阳阳”因为次数只有 5 次，被淘汰）
```

步骤 3：在导航引擎利用名称来自动提示时，通过用户已输入的字符，预先提示其最有可能输入的前几条名称。

由于我们按照频率排序后的自动提示树已经是按照用户的使用频率来更新的自动提示树，所以，在用户输入某一个字符时，只需要依次将 List 中的记录全部显示即可。

如：用户输入“阳”时，更新后的显示如下。

第一个字符串是“阳明路”；

第二个字符串是“阳明餐厅”。

整体的显示效果如图 9-5 所示，排在最前面显示的是用户输入最频繁的字符串，也是用户最需要的字符串。

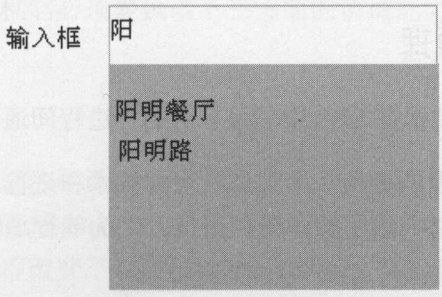


图 9-5　自动提示的最终效果

第 10 章

网络传输

网络传输是 LBS 的用户体验是否流畅的关键。一个 LBS 应用能否流畅，很大程度上取决于网络传输技术的合理运用。比如，以一个案例来说明：UC 的崛起，除了其在搜索框内网址的自动联想功能外，最主要的是其节省流量。在 2010 年，iPhone 在移动 2G 下用其他浏览器根本上不了网，客服通常也会推荐用 UC 浏览器，因为它节省流量。

可以猜想，UC 大致用到三方面的核心技术：自动联想提示、文字压缩和图像压缩。

- 自动联想提示功能与 9.3 节的“名称搜索”中所使用的技术类似，具体的使用可以参看名称搜索中的两个案例分析；
- 文字压缩：其方法有很多，比如 10.2 节中所讲述的文字压缩等；
- 图像压缩：是最简单的一种压缩方法，就是在 UC 的云端服务器对图像进行处理，把大图片变为小图片，之后再在 UC 浏览器客户端和云端服务器之间进行传输。当然，这种压缩依赖于 UC 必须经由 UC 服务器来获得所需的网页信息。

此外，减少网页传输的方法还有：预加载，以及让用户选择无图版来浏览网页。

总的来说，除计算机通信技术外，网络传输的关键技术还包括压缩技术和数据校验。

10.1 计算机通信原理

关于通信，有两种技术在 LBS 中是很有用的：进程间通信和网络通信。

进程间通信是多进程的基础，尽管多线程的技术和进程间通信稍有不同，但了解了进程间通信后，也就大致了解了线程通信，因为线程通信一般只用了进程通信中的同步技术。

网络通信是关于 TCP/IP 和 UDP 的知识。TCP/IP 是当今大多数 LBS 应用的网络通信协议；TCP/IP 通信的主流标准就是 Socket 通信；UDP 则往往会用于需要广播

的场合，比如流媒体。

10.1.1 进程间通信

本地的进程间通信（IPC）有很多种方式，通常可以总结为以下 4 类。

- 消息传递（管道、FIFO、消息队列）；
- 同步（互斥量、条件变量、读/写锁、记录锁、信号量）；
- 共享内存；
- 远程过程调用（Solaris 门和 Sun RPC）。

在实际的应用中，本地通信应用最多的是：消息传递、同步、共享内存。特别是共享内存（需结合同步技术使用），在多进程中应用最普遍。

1. 消息传递

在消息传递系统中，进程间的数据交换是以格式化的消息（Message）为单位的。若通信的进程之间不存在可直接访问的共享空间，则必须利用操作系统提供的消息传递方法实现进程通信。进程通过系统提供的发送消息和接收消息两个原语进行数据交换。

2. 管道通信

管道通信是消息传递的一种特殊方式。所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名 pipe 文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入（写）管道；而接收管道输出的接收进程（即读进程）则从管道中接收（读）数据。为了协调双方的通信，管道机制必须提供以下三方面的协调能力：互斥、同步和确定对方的存在。

3. 共享内存

共享内存是进程间通信中最简单的方式之一。共享内存允许两个或更多的进程访问同一块内存，就如同 malloc() 函数向不同的进程返回指向同一个物理内存区域的指针一样。当一个进程改变了这块地址中的内容时，其他进程都会察觉到这个更改。共享内存允许两个或多个进程共享同一块内存（这块内存会映射到各个进程自己独立的地址空间），从而使这些进程可以相互通信。

在 GNU/Linux 中，所有的进程都有唯一的虚拟地址空间，而共享内存应用编程接口 API 允许一个进程使用公共内存区段。但是对内存的共享访问的复杂度也相应地增加。共享内存的优点是简易性。

使用消息队列时，一个进程要向队列中写入消息，就会引起从用户地址空间向内核地址空间的一次复制，同样，一个进程进行消息读取时也要进行一次复制。共享内存的优点是完全省去了这些操作。共享内存会映射到进程的虚拟地址空间，进程对其可以直接访问，避免了数据的复制过程。

因此，共享内存是 GNU/Linux 现在可用的最快速的 IPC 机制。进程退出时会自动和已经挂接的共享内存区段分离，但是仍建议当进程不再使用共享区段时调用 shmdt 来卸载区段。

注意，当一个进程分支出父进程和子进程时，父进程先前创建的所有共享内存区段都会被子进程继承。如果区段已经做了删除标记（在前面以 IPC-RMID 指令调用 shmctl），而当前挂接数已经变为 0 的区段就会被移除。

（1）共享内存的使用流程

使用共享内存的流程如下。

1）进程首先必须分配共享内存。

2）随后需要访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的地址空间中。

3）当完成通信后，所有的进程都将脱离共享内存，并且由一个进程释放该共享内存块。

（2）共享内存分配所使用的重要函数

下面以 Linux 的 sharedmemory（共享内存）技术为例（Windows 下的共享内存技术是 filemapping 技术，但根本原理与 sharedmemory 相同），介绍共享内存分配所使用的重要函数如下。

- shmget( )：创建一个新的共享内存区段，取得一个共享内存区段的描述符；

- shmctl( )：取得一个共享内存区段的信息，为一个共享内存区段设置特定的信息，移除一个共享内存区段；
- shmat( )：挂接一个共享内存区段；
- shmdt( )：分离一个共享内存区段。

1）shmget 函数（分配）语句如下：

```
int segment_id = shmget (shm_key, int size , shmflag );
```

① 进程通过调用 shmget（Shared Memory GET，获取共享内存）函数来分配一个共享内存块。该函数的第一个参数是一个用来标识共享内存块的键值。

彼此无关的进程可以通过指定同一个键来获取对同一个共享内存块的访问。

不幸的是，其他程序也可能挑选了同样的特定值作为自己分配共享内存的键值，从而产生冲突。

用特殊常量 IPC_PRIVATE 作为键值可以保证系统建立一个全新的共享内存块。

② 该函数的第二个参数指定了所申请的内存块的大小。因为这些内存块是以页面为单位进行分配的，实际分配的内存块大小将被扩大到页面大小的整数倍。

③ 第三个参数是一组标志，通过特定常量的按位或操作来使用 shmget。这些特定的常量包括：

- IPC_CREAT：这个标志表示应创建一个新的共享内存块。通过指定该标志，可以创建一个具有指定键值的新共享内存块；
- IPC_EXCL：这个标志只能与 IPC_CREAT 同时使用。当指定该标志时，如果已有一个具有这个键值的共享内存块存在，则 shmget 会调用失败。也就是说，这个标志将使线程获得一个“独有”的共享内存块。如果没有指定这个标志，而系统中存在一个具有相同键值的共享内存块，shmget 会返回这个已经建立的共享内存块，而不是重新创建一个。

IPC_EXCL 标志的值由 9 个位组成，分别表示属主、属组和其他用户对该内存块的访问权限。其中，表示执行权限的位将被忽略。

指明访问权限的一个简单办法是利用<sys/stat.h>指定，例如：

S_IRUSR 和 S_IWUSR 分别指定该内存块属主的读写权限。

S_IROTH 和 S_IWOTH 则指定其他用户的读写权限。

2）shmat（Shared Memory Attach，绑定）函数语句如下：

```
pst= shmat(iShm_id, NULL, 0)
```

一个进程获取对一块共享内存的访问，这个进程必须先调用 shmat（SHared Memory Attach，绑定到共享内存）。

① 将 shmget 返回的共享内存标识符 SHMID 传递给这个函数作为第一个参数。

② 第二个参数是一个指针，指向你希望用于映射该共享内存块的进程内存地址；如果指定 NULL，则 Linux 会自动选择一个合适的地址用于映射。

③ 第三个参数是一个标志位，包含以下选项。

- SHM_RND：表示第二个参数指定的地址应被向下靠拢到内存页面大小的整数倍。如果不指定这个标志，则在调用 shmat 时必须手工将共享内存块的大小按页面大小对齐；
- SHM_RDONLY：表示这个内存块仅允许读取操作，而禁止写入。如果这个函数调用成功，则会返回绑定的共享内存块对应的地址。

3）shmdt 函数（Shared Memory Detach，脱离共享内存块）。

通过 fork 函数创建的子进程将可以继承父进程的共享内存块；如果需要，它们可以主动脱离这些共享内存块。当一个进程不再使用一个共享内存块时，应通过调用 shmdt 函数。

如果释放这个内存块的进程是最后一个使用该内存块的进程，则这个内存块将被删除。

对 exit 或任何 exec 族函数的调用都会自动使进程脱离共享内存块。

4）shmctl 函数（控制和释放）语句如下：

```
shmctl(iShm_id,IPC_RMID,0)<0
```

调用 shmctl（Shared Memory Control，控制共享内存）函数会返回一个共享内存

块的相关信息，同时 shmctl 允许程序修改这些信息。

该函数的第一个参数是一个共享内存块标识。要获取一个共享内存块的相关信息，则为该函数传递 IPC_STAT 作为第二个参数，同时传递一个指向 struct shmid_ds 对象的指针作为第三个参数。

要删除一个共享内存块，则应将 IPC_RMID 作为第二个参数，而将 NULL 作为第三个参数。当最后一个绑定该共享内存块的进程与其脱离时，该共享内存块将被删除。

应当在结束使用每个共享内存块时都使用 shmctl 进行释放，以防止超过系统所允许的共享内存块的总数限制。调用 exit 和 exec 会使进程脱离共享内存块，但不会删除这个内存块。

共享内存的总体大小是有限制的，这个大小通过 SHMMAX 参数来定义（以字节为单位），可以通过执行以下命令来确定 SHMMAX 的值：

```
cat /proc/sys/kernel/shmmax
```

修改共享内存：

```
设置 SHMMAX
# >echo "2147488364" > /proc/sys/kernel/shmmax
```

也可以使用 sysctl 命令来更改 SHMMAX 的值：

```
# sysctl -w kernel.shmmax=2147488364
```

最后，通过将该内核参数插入到 /etc/sysctl.conf 启动文件中，使这种更改永久有效：

```
# echo "kernel.shmmax=2147488364" >> /etc/sysctl.conf
```

（3）共享内存用法实例

该实例的功能是：创建一个共享内存区段，并显示其相关信息，然后删除该内存共享区。

```
1  #include <stdio.h>
2  #include <unistd.h>  //getpagesize
3  #include <sys/ipc.h>
4  #include <sys/shm.h>
```

```
#define MY_SHM_ID 19801130
int main( )
    {
        //获得系统中页面的大小
        printf( "page size=%d/n",getpagesize( ) );
        //创建一个共享内存区段
        int shmid,ret;
        shmid=shmget( MY_SHM_ID,4096,0666|IPC_CREAT );
        //创建一个 4KB 大小的共享内存区段。指定的大小必须是当前系统架构中
        //页面大小的整数倍
        if( shmid>0 )
            printf( "Create a shared memory segment %d/n",shmid );
        //获得一个内存区段的信息
        struct shmid_ds shmds;
        //shmid=shmget( MY_SHM_ID,0,0 );//示例怎样获得一个共享内存的标识符
        ret=shmctl( shmid,IPC_STAT,&shmds );
        if( ret==0 )
            {
              printf( "Size of memory segment is %d/n",shmds.shm_segsz );
              printf( "Numbre of attaches %d/n",( int )shmds.shm_nattch );
            }
        else
            {
                printf( "shmctl( ) call failed/n" );
            }
        //删除该共享内存区
        ret=shmctl( shmid,IPC_RMID,0 );
        if( ret==0 )
            printf( "Shared memory removed /n" );
        else
            printf( "Shared memory remove failed /n" );
        return 0;
    }
```

（4）共享内存总结

共享内存块提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取或写入数据，但所有的程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现。需注意的是，Linux 无法严格保证提供对共享内存块的独占访问，甚至是在通过使用 IPC_PRIVATE 创建新的共享内存块时也不能保证访问的独占性。 同时，多个使用共享内存块的进程之间必须协

调使用同一个键值。

10.1.2 网络通信

1. TCP/IP 通信

TCP/IP 协议栈分为四层：链路层、网络层、运输层和应用层，其中，主要特指运输层的通信协议，如图 10-1 所示。

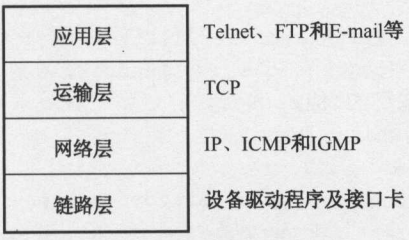


图 10-1

在网络通信中，由于 UNIX 占据统治地位，所以，TCP/IP 所用的 Socket 技术是最主流的网络通信技术。

TCP 和 IP 就像信封一样，要传递的信息被划分成若干段，每一段塞入一个 TCP 信封，并在该信封面上记录有分段号的信息，再将 TCP 信封塞入 IP 大信封，发送上网。在接收端，一个 TCP 软件包收集信封，抽出数据，按发送前的顺序还原，并加以校验，若发现差错，TCP 将会要求重发。因此，TCP/IP 在 Internet 中几乎可以无差错地传送数据。在任何一个物理网络中，各站点都有一个机器可识别的地址，该地址叫作物理地址。

TCP 头包括源端口（16 位）、目标端口（16 位）、序列号（32 位，确保数据到达序列正确的编号）、应答号（32 位，期望的下一个 TCP 数据段）、头长度（4 位，以 32 位字长为单位的报头长度）、保留（6 位，置为 0）、编码位（6 位，开始终止会话类的控制功能）、窗口（16 位，用来控制流量）、校验和（16 位，头标和数据域计算的校验和）、紧急（16 位，指示紧急数据的末端）、可选项（如果有 0 或 32 位，TCP 段的最大值）、数据（可变位数，上层协议的数据）。

一个 TCP/IP（即 Socket 技术）的代码示例如下：

```
1.        //服务器端 Socket 程序
```

```
2.          public void ServerSocket()
3.          {
4.            Socket listener = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);
          //(1)在服务器端定义一个套接字(socket),使用 TCP 协议
5.              listener.Bind(new IPEndPoint(IPAddress.Any, 1980));
          //(2)设定该套接字的绑定 IP
6.              listener.Listen(200);
          //(3)将套接字置为监听状态，并设置监听队列为 100
7.              while (true)
8.              {
9.                  string receiverAllStr = string.Empty;
10.                 Socket socket = listener.Accept();
          //(4)为新连接建立新的 socket
11.                 while (true)
12.                 {
13.                     byte[] receiveBytes = new byte[2048];
14.                     int numBytes = socket.Receive(receiveBytes);
          //(5)从 Socket 中接收消息，将接收到的消息写入已定义的字节数组中
15.                     string receiveStr = Encoding.ASCII.GetString(rec
          eiveBytes, 0, numBytes);
16.                     receiverAllStr += receiveStr;
17.                     if (receiveStr.IndexOf("[FINAL]") > -1)
18.                     {
19.                         Console.WriteLine(receiverAllStr);
20.                         break;
21.                     }
22.                 }
23.                 string replySuccess = "The data can be successfullly
                                          received!";
24.                 byte[] byteStr = Encoding.ASCII.GetBytes (replySuccess);
          //非必需
25.                 socket.Send(byteStr);
          //非必需，给客户端发送一个接收成功的回复 send data to connected System.Net. Sockets.Socket
26.                 socket.Shutdown(SocketShutdown.Both);
27.                 socket.Close();
          //(6)关闭 socket Connection，释放所有已占用资源，在死循环内继续接受其他消息
28.             }
29.             listener.Close();
30.             Console.Read();
31.         }
32.         //客户端的 socket 程序 新建一个 main 程序来运行发送程序
33.         public void ClientSocket()
```

```
34.      {
35.          Socket sender = new Socket(AddressFamily.InterNetwork, S
                              ocketType.Stream, ProtocolType.Tcp);
       //(1)在服务器端定义一个套接字(socket),使用 TCP 协议
36.          IPHostEntry ipHost=Dns.Resolve("127.0.0.1");
37.          IPAddress ipAddress=ipHost.AddressList[0];
38.          IPEndPoint ipEndPoint=new IPEndPoint(ipAddress,2010);
39.          sender.Connect(ipEndPoint);
       //(2)设定该套接字的链接 IP，发送到指定 IP 地址的 2010 端口
40.          string sendStr = "socket test.";
41.          byte[] sendBytes = Encoding.ASCII.GetBytes(sendStr+"
                                    [FINAL]");
42.          sender.Send(sendBytes);//(3)发送数据比特流
43.          byte[] receiveBytes=new byte[1024];
44.          sender.Receive(receiveBytes);//(4)获取服务器的响应数据
45.          Console.WriteLine("received from server: " + Encoding.
              ASCII.GetString(receiveBytes));
46.          sender.Shutdown(SocketShutdown.Both);
47.          sender.Close();
       //(5)关闭 socket Connection，释放所有已占用资源
48.      }
```

网络通信主要就是要考虑效率问题，而 TCP/IP（Socket 协议）已经成为目前网络通信的主要标准。使用 Socket 的主要方法有以下几种：select（一种轮询查找）、poll（线程池）、epoll（或 IOCP）。主要的区别是：对于 select/poll 方式，进程只有与内核进行一定的复制操作后，内核才会对所有监视的文件描述符进行扫描；而 epoll 事件没有内核复制的耗时操作，通过 epoll_ctl()注册一个文件描述符，一旦某个文件描述符就绪，内核会采用类似 call back 的回调机制，迅速激活这个文件描述符，epoll_wait()便会得到通知。具体的机制将在第 11 章介绍。

2. UDP 协议

UDP 是不可靠或无连接分组交付的网络通信协议，可认为是一种运输层的通信协议，不管是否接收到，如同广播喇叭一样。

因为 UDP 协议速度很快，且具有广播的功能，所以流媒体应用一般用 UDP，需要用到广播技术的应用时，也必须用 UDP 协议。

UDP 协议提供了一种高效可靠的网络上传输数据的通信方式，同时，UDP 不用消耗不必要的网络资源和处理时间。使用 UDP 协议包括 TFTP、SNMP、NFS、DNS、DHCP。很适合客户机向服务器发送简单服务请求的环境，因为开销比 TCP 小，TCP

要经历三次握手过程。UDP 依靠上层协议提供可靠性，包括处理报文的丢失、重复、时延、乱序、连接失效等问题。

UDP 头包括源端口、目标端口号、报文头长度（8 位组 UDP 数据包）、校验和（头标和数据域计算的校验和，在高可靠性网络上尽量减少开销）、数据（上层协议数据）。

下面介绍 UDP 应用的一个示例，功能如下。

1）利用 Socket 编程建立服务器与客户端之间的 UDP 连接。

2）完成 C/S 之间的数据通信，包括以下两点。

① Client 通过给定的 Server IP 和端口号向 Server 端发送请求。请求来自 cmd 命令行用户输入。

② Server 端保持监听状态，收到 Client 端请求后，将请求内容输出到命令行。

服务器端代码如下：

```
// 服务器与客户端通信的函数
//如果服务器接收到客户端的请求 ，将打印接收到的信息，并且回复："OK".
//服务器
#include "stdio.h"
#include "Winsock2.h"
#include "iostream"
#include "string"


#define HOST_IP 127.0.0.1
#define HOST_PORT 8080
#define OK_STR "OK"

void main(){
    //Winsock 的版本
    int version_a = 1;//低位
    int version_b = 1;//高位


    WORD versionRequest = MAKEWORD(version_a,version_b);
    WSAData wsaData;
```

```
24.     int err;
25.
26.     err = WSAStartup(versionRequest, &wsaData);
27.
28.     if(err != 0 ){
29.         printf("ERROR!");
30.         return;
31.     }
32.
33.     if (LOBYTE(wsaData.wVersion)  != 1 || HIBYTE(wsaData.wVersion) !
                                                          = 1)
34.     {
35.         printf("WRONG WINSOCK VERSION!");
36.         WSACleanup();
37.         return;
38.     }
39.
40.     /*
41.     *建立 socket
42.     *第一个参数是协议类型，通常为 IP 网络使用 AF_INET。
43.     *第二个参数是 socket 类型，对 UDP 来说，使用 SOCK_DGRAM；对 TCP 来说，使用
          *SOCK_STREAM。
44.     *最后一个参数是通信协议，通常是 0。
45.     */
46.     SOCKET socServer = socket(AF_INET, SOCK_DGRAM,0);
47.
48.     SOCKADDR_IN addr_Srv;
49.
50.
51.     addr_Srv.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
52.
53.     addr_Srv.sin_family = AF_INET;
54.     addr_Srv.sin_port = htons(HOST_PORT);
55.
56.     // 将 socket 与地址绑定
57.     bind(socServer,( SOCKADDR*) &addr_Srv, sizeof(SOCKADDR));
58.
59.     //客户端的地址
60.     SOCKADDR_IN addr_Clt;
61.     char recvBuf[100];
62.
63.     int fromlen = sizeof(SOCKADDR);
64.
65.     while(true){
```

```
66.          //从服务器中接收数据
67.          recvfrom(socServer, recvBuf, 100, 0,  (SOCKADDR*) &addr_Clt,
                    &fromlen);
68.          //向命令行输出数据
69.          std::cout<<recvBuf<<std::endl;
70.          // 服务器向客户端发送“OK”表示已经接收到数据
71.          .
72.
73.          sendto(socServer, OK_STR, strlen(OK_STR)+1, 0, (SOCKADDR*)&
                    addr_Clt, sizeof(SOCKADDR));
74.      }
75.
76.      // 最后关闭 Socket
77.      closesocket(socServer);
78.
79.      WSACleanup();}
```

客户端代码如下：

```
1.  // 客户端与服务器端通信的函数
2.
3.  //客户端
4.  #include "Winsock2.h"
5.  #include "iostream"
6.  #include "stdio.h"
7.  #pragma comment(lib, "ws2_32.lib")
8.
9.
10. #define HOST_IP "127.0.0.1"
11. #define HOST_PORT 8080
12.
13. void main(){
14.      // Winsock 的版本
15.      int version_a = 1;//低位
16.      int version_b = 1;//高位
17.
18.
19.      WORD versionRequest = MAKEWORD(version_a,version_b);
20.      WSAData wsaData;
21.      int error;
22.      error = WSAStartup(versionRequest, &wsaData);
23.
24.      if(error != 0    ){
25.          printf("ERROR!");
```

```
26.         return;
27.     }
28.
29.     if (LOBYTE(wsaData.wVersion)  != 1 || HIBYTE(wsaData.wVersion) !
            = 1)
30.     {
31.         printf("WRONG WINSOCK VERSION!");
32.         WSACleanup();
33.         return;
34.     }
35.
36.
37.     //制作请求
38.     char requestStr[100];
39.
40.     //建立 socket
41.     SOCKET socClient = socket(AF_INET, SOCK_DGRAM, 0);
42.     SOCKADDR_IN addrSrv;
43.     addrSrv.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
44.     addrSrv.sin_family=AF_INET;
45.     addrSrv.sin_port=htons(HOST_PORT);
46.
47.     // 存储服务器返回的数据
48.     char cRecvBuf[100];
49.
50.
51.
52.     while(true){
53.
54.         std::cin>>requestStr;
55.         // 向服务器发送请求
56.         sendto(socClient, requestStr, strlen(requestStr)+1, 0, (SOCK
                    ADDR*) &addrSrv, sizeof(SOCKADDR));
57.
58.         //接收服务器发来的反馈
59.
60.         recv(socClient, cRecvBuf, strlen(cRecvBuf)+1, 0);
61.
62.         std::cout<< cRecvBuf << std::endl;
63.     }
64.
65.     //关闭 socket，并清除 wsadata
66.     closesocket(socClient);
67.     WSACleanup();
```

```
68.
69.
70.}
```

10.2 压缩算法

压缩算法往往会用于数据库、图片或文件的压缩，以减少硬盘或内存的占有量，或者网络数据在传输时的压缩以减少网络的传输量，提高网络的传输速度。

具体的压缩算法除了傅里叶变换或者滤波外，还有多种文字压缩的方法。

1. 字典算法

字典算法是最简单的压缩算法之一。该算法把文本中出现频率比较多的单词或词汇组合做成一个对应的字典列表，并用特殊代码来表示这个单词或词汇。

例如，字典列表如下：

```
00=Chinese
01=People
02=China
```

源文本为：

```
I am a Chinese people,I am from China
```

压缩后的编码为：I am a 00 01，I am from 02。

压缩编码后的长度显著缩小。

2. 固定位长算法（Fixed Bit Length Packing）

固定位长算法也是比较常见的压缩算法之一。这种算法是把文本用需要的最少位来进行编码压缩。

比如，8 个十六进制数为：1、2、3、4、5、6、7、8。

转换为二进制数：00000001、00000010、00000011、00000100、00000101、00000110、00000111、00001000。

每个数只用到了低 4 位，而高 4 位没有用到（全为 0），因此，对低 4 位进行压缩编码后得到：0001、0010、0011、0100、0101、0110、0111、1000。然后补充为

字节得到：00010010、00110100、01010110、01111000。

所以，原来的 8 个十六进制数缩短了一半，得到 4 个十六进制数：12、34、56、78。

3. RLE 算法

这种压缩编码是一种变长的编码，RLE 根据文本的不同会有不同的压缩编码变体与之相适应，以产生更大的压缩比率。

变体 1：重复次数+字符

文本字符串：A A A B B B C C C C D D D D

编码后得到：3 A 3 B 4 C 4 D。

变体 2：特殊字符+重复次数+字符

文本字符串：A A A A A B C C C C B C C C

编码后得到：B B 5 A B B 4 C B B 3 C。

编码串的最开始说明特殊字符 B B，B B 后面跟着的数字表示重复的次数。

变体 3：把文本中的每字节分组成块，每个字符最多重复 127 次。每个块以一个特殊字节开头，那个特殊字节的第 7 位如果被置位，则剩下的 7 位数值就是后面字符的重复次数。如果第 7 位没有被置位，则剩下的 7 位就是后面没有被压缩的字符数量。

例如，文本字符串为：A A A A A B C D E F F F，编码后得到：85 A 4 B C D E 83 F（85H= 10000101B、4H= 00000100B、83H= 10000011B）。

以上 3 种 RLE 的变体是最常用的，还有很多其他的变体算法，这些算法在 Winzip、WinRAR 软件中也是经常用到的。

4. LZ77 算法

LZ77 算法是由 Lempel-Ziv 在 1977 年发明的，也是 GBA 内置的压缩算法。LZ77 算法有许多派生算法（其中包括 LZSS 算法）。它们的原理基本相同，无论是哪种派生算法，LZ77 算法总会包含一个动态窗口（Sliding Window）和一个预读缓冲器（Read Ahead Buffer）。动态窗口是一个历史缓冲器，它被用来存放输入流的前 n 字

节的有关信息。一个动态窗口的数据范围可以从 0 字节到 64KB，而 LZSS 算法使用了 4KB 的一个动态窗口。预读缓冲器是与动态窗口相对应的，预读缓冲器的大小通常在 0～258 之间。用输入流的后 *k* 字节填充预读缓存器（这里的 *k* 是预读缓存器的大小）。在动态窗口中寻找与预读缓冲器中最匹配的数据，如果匹配的数据长度大于最小匹配长度（通常取决于编码器和动态窗口的大小，比如，一个 4KB 的动态窗口，它的最小匹配长度是 2），那么就输出一对<长度（length），距离（distance）>数组。长度（length）是匹配的数据长度，而距离（distance）代表在动态窗口中向后多少字节可以得到要匹配的数据。动态窗口的数据随之向后增加长度（length）字节。

例如：假设有一个 10 字节的动态窗口和一个 5 字节的预读缓存器。

文本：A A A A A A A A A A A B A B A A A A A

其中，动态窗口：A A A A A A A A A A ；

预读缓存器：B A B A A。

动态窗口中包含 10 个 A ，这就是最后读取的 10 字节。预读缓存器中包含了 B A B A A。

编码的第一步就是寻找动态窗口与预读缓存器相似长度大于 2 的字节部分。在动态窗口中找不到 B A B A A，所以，B 就被按照字面输出。然后动态窗口增加了 1 字节，现在开始增加了一个 B。

第二步：A A A A A A A A A A A B A B A A A A A

其中，动态窗口： A A A A A A A A A AB；

预读缓存器：A B A A A。

现在预读缓存器包含 A B A A A，然后与动态窗口进行比较。这时，在动态窗口找到了相似长度为 2 的 A B，因此，一对<长度，距离>就被输出了，长度（length）是 2，并且向后的距离是 9，所以输出为<2,9>，然后动态窗口滑过 2 字节。现在已经输出了 B <2,9>。

第三步：A A A A A A A A A A A B A B A A A A A

继续上面的方法得到输出结果<5,0>。现在已经输出了 B <2,9> <5,0>。最终的编码结果是：A A A A A A A A A A A B <2,9> <5,0>。

由于数组是无法直接用二进制数来表示的，LZ77 会把编码的每八个数分成一组，每组前用一个前缀标识来说明这八个数的属性。比如数据流：A B A C A C B A C A 按照 LZ77 的算法编码为：A B A C<2,2> <4,1>，刚好八个数。按照 LZ77 的规则，用“0”表示原文输出，“1”表示数组输出。所以，这段编码就表示为：00001111B（等于 0FH），因此，得到完整的压缩编码表示：F A B A C 2 2 4 1。虽然表面上只缩短了 1 字节的空间，但当数据流很长时，就会突出它的优势，这种算法在 ZIP 格式中会经常用到。

5. 霍夫曼编码（Huffman Encoding）

霍夫曼编码是通过字符出现频率、优先队列（Priority Queue）和二叉树来进行的一种压缩算法，这种二叉树又叫 Huffman 二叉树或最优二叉树，即一种道路带权路径最小的二叉树。

例如，如果需要压缩下面的字符串：

```
“beep boop beer!”
```

首先，计算出每个字符出现的次数，得到如图 10-2 所示的字符次数统计表。

字符	次数
'b'	3
'e'	4
'p'	2
' '	2
'o'	2
'r'	1
'!'	1

图 10-2　字符次数统计表

然后，把这些字符和统计次数放到优先队列中（用出现的次数作为优先的依据）。我们可以看到，优先队列是以统计次数为依据排序一个数组，如果统计次数一样，就会使用出现的次序排序。图 10-3 是我们得到的优先队列。

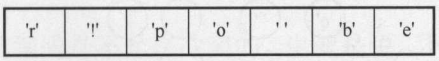


图 10-3　优先队列

接下来把这个优先队列转成二叉树。我们采取自底而上构建优先队列的方法，

始终从队列的头取两个元素来构造一个二叉树，把第一个元素作为左结点，第二个元素作为右结点，并把这两个元素的统计次数相加后放回优先队列中，然后得到如图 10-4 所示的数据表。

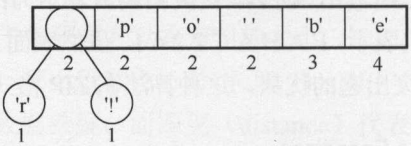


图 10-4　第一次组织的优先队列

接着，把前两个取出来形成一个统计次数为 2+2=4 的结点，然后放回优先队列中，得到如图 10-5 所示的优先队列。

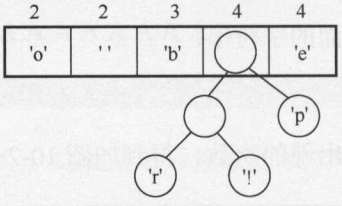


图 10-5　第二次组织的优先队列

继续处理优先队列，依次得到如图 10-6 至图 10-8 所示的优先队列。

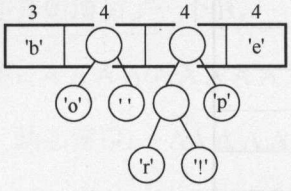


图 10-6　第三次组织的优先队列

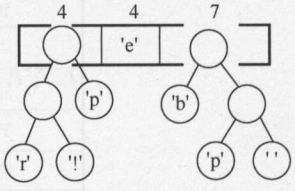


图 10-7　第四次组织的优先队列

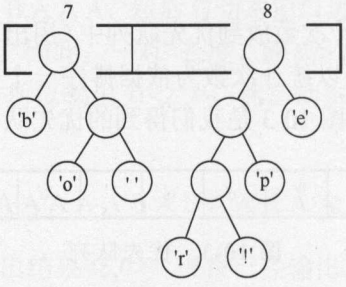


图 10-8　第五次组织的优先队列

最终得到如图 10-9 所示的一棵二叉树。

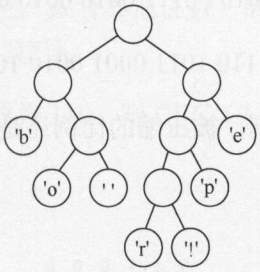


图 10-9　最终的优先队列

此时，我们把这棵树的左支编码为 0，右支编码为 1，这样就可以遍历这棵树得到字符的编码，比如：'b'的编码是 00，'p'的编码是 101，'r'的编码是 1000。我们可以看到，出现频率越多的，就越在上层，编码也越短，出现频率越少的，就越在下层，编码也越长。

最终可以得到如图 10-10 所示的霍夫曼树和如图 10-11 所示的编码表。

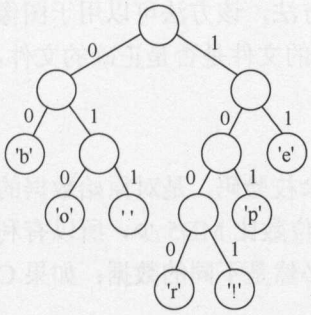


图 10-10　最终的霍夫曼树

字符	编码
'b'	00
'e'	11
'p'	101
' '	011
'o'	010
'r'	1000
'!'	1001

图 10-11　霍夫曼编码表

当编码的时候，需按“bit”进行，解码时也是通过 bit 来完成的。例如：bitset 如果为“1011110111”，那么其解码后就是“pepe”。所以，需要通过这个二叉树建立 Huffman 编码和解码的字典表。

Huffman 对各字符的编码是不允许冲突的。也就是说，不会存在某一个编码是另一个编码的前缀，因为编码后的字符是没有分隔符的。

于是，对原始字符串 beep boop beer!，则其对应的二进制数为：0110 0010 0110

0101 0110 0101 0111 0000 0010 0000 0110 0010 0110 1111 0110 1111 0111 0000 0010 0000 0110 0010 0110 0101 0110 0101 0111 0010 0010 0001。

Huffman 的编码为：0011 1110 1011 0001 0010 1010 1100 1111 1000 1001。

从这个例子中我们可以看到，被压缩的比例还是很可观的。

10.3 数据检验

一般而言，数据校验有以下两个用途。

- 网络数据传输完成后，为了保证不会因丢包导致数据错误而进行的一种校验方法。比如：可以保证下载的应用程序能使用。
- 在对两个数据进行对比时，能快速找到相同或不相同的数据。可用于图像的对比、地图数据的对比等场合。

数据检验有两种常用的方法：MD5 方法和 CRC 方法。

1. MD5

MD5 一种对原始数据的抽取 128 位特征的方法，该方法可以用于图像等大数据的对比，比如两个软件包的对比，或者检查下载的文件是否是正确的文件。

2. CRC

CRC（Cyclic Redundancy Check）即循环冗余校验码，是对原始数据的抽取特征（一般小于 32 位）的一种方法。该方法由于特征位数比 MD5 少，所以有利于快速对数据进行对比。如果两个数据的 CRC 不同，则必然是不同的数据；如果 CRC 相同，再进一步对数据的每一位进行对比。

一般来说，CRC 的方法在 LBS 应用中比 MD5 更为常见，这是因为 CRC 往往是信息传输中实用的快速验证方法。

CRC 的一种使用场景如下：在发送端根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的 r 位监督码（CRC 码），附在原始信息后边，构成一个新的二进制码序列数共 $k+r$ 位，然后发送出去。在接收端，根据信息码和 CRC 码之间所遵循的规则进行检验，以确定传送中是否出错。这个规则在差错控制理论中称为“生成多项式”。

下面具体介绍 CRC 的原理：对一个数据，先把它转换为二进制数据，之后将它除以一个 32 位（也可以是 26 位，或者其他位数）的值，其余数就是 CRC。这个余数就是我们所要求的原来数据的特征。

假设数据为：100110，除数为 110，则 CRC（3 位监督码）为 010。

具体的计算过程如下：

```
          0 0 0 1 1 0      商
      ┌──────────────
1 1 0 ) 1 0 0 1 1 0
        1 1 0
      ──────────
          0 1 1 1
            1 1 0
          ──────
              1 0    余数
```

第 11 章

后台服务

现在的 LBS 应用一般是基于网络服务的应用，所以需要与服务器之间通信（即 Web Service），能处理高并发请求，为了提高网络请求 I/O 的处理速度，还需要具备多线程、多进程的架构。

11.1 Web Service

网络通信通常可以选择：Socket、Web Service 和 Json。由于 Socket 是底层技术，开发成本较高，而 Json 的应用往往局限于浏览器，所以，Web Service 已经成为目前 LBS 应用与服务器通信的主流通信机制。

1. Web Service 基本概念

Web Service 也叫 XML Web Service，是使用 SOAP 协议实现跨编程语言和跨操作系统平台的。

Web Service 是一种可以接收“从 Internet 或者 Intranet 上的其他系统中传递过来的请求”的轻量级的通信技术。

2. Web Service 和 Socket 的比较

Socket 和 Web Service 都有跨平台的优点，但是 Socket 偏底层，效率高，而且开发成本大。

Socket 提供了 TCP/IP 或者 UDP 的通信实现，如果做标准的服务器，比如下载服务器、语音通信的程序、视频、文件传输等只能用 Socket。

Web Service 基于 TCP/IP 协议，是以 XML 为载体的通信方式。Web Service 效率低，但是开发成本低廉，而且由于其很直观，格式很标准，所以也方便于异构系统的交互。

在 LBS 服务中，如果想提供对外的业务访问的接口，可能需要支持浏览器、各种语言的客户端等各种异构系统，在这种情况下，Web Service 是首选。

3. Web Service 和 JSON 的比较

JSON 从本质说就是 JavaScript 片段描述对象，对浏览器会很有用。比如：如果要在浏览器上做一个 AJAX 的功能，JSON 是最简单有效的选择。实际上，电影网站 mtime 的很多异步功能都是用 JSON 方式传递的。

JSONlib 等库都可以直接把一个对象转为 JSON 字符串，struts2 也提供了 JSON 插件。 简单的应用特别是数据库方面的添加、删除等操作，使用 JSON 的方式是可以的。

JSON 没有什么门槛，也可以用于异构系统交互，不过，如果异构系统不仅仅是浏览器，那还是选择 Web Service。这是由于 Web Service 在安全性、接口标准性等方面都远远优于 JSON。

4. 远程调用技术的作用

Web Service 其实是一种跨编程语言和跨操作系统平台的远程调用技术。所谓远程调用，就是一台计算机 a 上的一个程序可以调用到另外一台计算机 b 上的一个对象的方法。例如，银联提供给商场的 POS 刷卡系统。

- 远程调用技术有什么用呢？远程调用技术可以使商场的 POS 机转账调用的转账方法的代码存在银行服务器上，而不是商场的 POS 机上。
- 什么情况下可能用到远程调用技术？例如，Amazon、天气预报系统、淘宝网、校内网、百度等把自己的系统服务以 Web Service 服务的形式暴露出来，让第三方网站和程序可以调用这些服务功能，这样扩展了自己系统的市场占有率，也就是所谓的 SOA（面向服务的体系结构）应用。

Web Service 可以跨编程语言和跨操作平台，就是说服务器端程序采用 Java 编写，客户端程序则可以采用其他编程语言编写，反之亦然。跨操作系统平台则是指服务器端程序和客户端程序可以在不同的操作系统上运行。

除 Web Service 外，常见的远程调用技术还有 RMI（Remote method invoke）和 CORBA，由于 Web Service 具有跨平台和跨编程语言的特点，因此，比其他两种技术应用更广泛，但性能略低。

5. Web Service 的优点

Web Service 的主要目标是跨平台的可互操作性。为了实现这一目标，Web Service

完全基于 XML（可扩展标记语言）、XSD（XML Schema）等独立于平台和软件供应商的标准，成为创建可互操作的、分布式应用程序的新平台。使用 Web Service 的优点如下。

（1）跨防火墙的通信

如果应用程序有成千上万的用户，而且分布在世界各地，那么客户端和服务器之间的通信将是一个棘手的问题。因为客户端和服务器之间通常会有防火墙或者代理服务器。传统的做法是，面对这种跨防火墙的通信，选择用浏览器作为客户端，写一大堆 HTML 页面（如 ASP 语言开发），等于把应用程序的中间层暴露给最终用户。这样做的结果是开发难度大，程序很难维护。要是客户端代码不再如此依赖于 HTML 表单，客户端的编程就简单多了。如果中间层组件换成 Web Service，就可以从用户界面直接调用中间层组件，从而省去建立 HTML 页面的那一步。要调用 Web Service，可以直接使用 Microsoft SOAP Toolkit 或.NET 这样的 SOAP 客户端，也可以使用自己开发的 SOAP 客户端，然后把它和应用程序连接起来。不仅缩短了开发周期，还减少了代码的复杂度，并能够增强应用程序的可维护性。

（2）应用程序集成

企业级的应用程序开发者都知道，企业经常都要把用不同语言写成的、在不同平台上运行的各种程序集成起来，而这种集成将花费很大的开发力量。应用程序经常需要从运行的一台主机上的程序中获取数据；或者把数据发送到主机或其他平台应用程序中。即使在同一个平台上，不同软件厂商生产的各种软件也常常需要集成起来。通过 Web Service，应用程序可以用标准的方法把功能和数据“暴露”出来，供其他应用程序使用。

XML Web services 提供了在松耦合环境中使用标准协议（HTTP、XML、SOAP 和 WSDL）交换消息的能力。消息可以是结构化的、带类型的，也可以是松散定义的。

（3）B2B 的集成

B2B（Business to Business）是指商家（泛指企业）对商家的电子商务，即企业与企业之间通过互联网进行产品、服务及信息的交换。通俗的说法是指进行电子商务交易的供需双方都是商家（或企业、公司），它们使用了 Internet 技术或各种商务网络平台，完成商务交易的过程。

Web Service 是 B2B 集成成功的关键。通过 Web Service，公司只需把关键的商

务应用“暴露”给指定的供应商和客户即可。Web Service 运行在 Internet 上，在世界的任何地方都可轻易实现，其运行成本就相对较低。用 Web Service 来实现 B2B 集成的最大好处在于可以轻易实现互操作性。只要把商务逻辑“暴露”出来，成为 Web Service，就可以让任何指定的合作伙伴调用这些商务逻辑，而不管其系统在什么平台上运行，使用什么开发语言，这样就大大减少了花在 B2B 集成上的时间和成本。

（4）软件和数据重用

Web Service 在允许重用代码的同时，也可以重用代码背后的数据。使用 Web Service，再也不必像以前那样，要先从第三方购买、安装软件组件，再从应用程序中调用这些组件；只需要直接调用远端的 Web Service 即可。另一种软件重用的情况是，把多个应用程序的功能集成起来，通过 Web Service “暴露”出来，就可以非常容易地把所有的这些功能都集成到你的门户站点中，为用户提供一个统一、友好的界面。 可以在应用程序中使用第三方的 Web Service 提供的功能，也可以把自己的应用程序功能通过 Web Service 提供给他人。两种情况下都可以重用代码和代码背后的数据。

6. Web Service 的缺点

虽然 Web Service 在通过 Web 进行互操作或远程调用时是最有用的。不过，也有一些情况下，Web Service 根本不能带来任何好处。

（1）单机应用程序

目前，企业和个人还在使用很多桌面应用程序，其中一些只需要与本机上的其他程序通信。在这种情况下，最好不要用 Web Service，只要用本地的 API 就可以。COM 非常适合在这种情况下工作，因为它既小又快。运行在同一台服务器上的服务器软件也是这样。当然，Web Service 也能用在这些场合，但那样不仅消耗太大，而且不会带来任何好处。

（2）局域网的一些应用程序

在许多应用中，所有的程序都是在 Windows 平台下使用 COM，都运行在同一个局域网中。在这些程序里，使用 DCOM 会比 SOAP/HTTP 有效得多。与此类似，如果一个.NET 程序要连接到局域网上的另一个.NET 程序，应该使用．NET Remoting。其实在．NET Remoting 中也可以指定使用 SOAP/HTTP 进行 Web Service 调用。不过最好还是直接通过 TCP 进行 RPC 调用，那样会有效得多。

7. Web Service 的调用原理

Web Service 通过 SOAP 在 Web 上提供的软件服务，使用 WSDL 文件进行说明，并通过 UDDI 进行注册。

Web 服务有以下两层含义。

- 指封装成单个实体并发布到网络上的功能集合体；
- 指功能集合体被调用后所提供的服务。

简单地讲，Web 服务是一个 URL 资源，客户端可以通过编程方式请求得到它的服务，而不需要知道所请求的服务是怎样实现的，这一点与传统的分布式组件对象模型不同。

Web 服务的体系结构是基于 Web 服务提供者、Web 服务请求者、Web 服务中介者三个角色和发布、发现、绑定三个动作构建的。简单地说，Web 服务提供者就是 Web 服务的拥有者，它耐心等待为其他服务和用户提供自己已有的功能；Web 服务请求者就是 Web 服务功能的使用者，它利用 SOAP 消息向 Web 服务提供者发送请求以获得服务；Web 服务中介者的作用是把一个 Web 服务请求者与合适的 Web 服务提供者联系在一起，它充当管理者的角色，一般是 UDDI。这三个角色是根据逻辑关系划分的，在实际应用中，角色之间很可能有交叉：一个 Web 服务既可以是 Web 服务提供者，也可以是 Web 服务请求者，或者二者兼而有之。显示了 Web 服务角色之间的关系，其中，“发布”是为了让用户或其他服务知道某个 Web 服务的存在和相关信息；“查找（发现）”是为了找到合适的 Web 服务；“绑定”则是在提供者与请求者之间建立某种联系。

如图 11-1 所示，实现一个完整的 Web 服务包括以下步骤。

1）Web 服务提供者设计实现 Web 服务，并将调试正确后的 Web 服务通过 Web 服务中介者发布，然后在 UDDI 注册中心注册；（发布）

2）Web 服务请求者向 Web 服务中介者请求特定的服务，中介者根据请求查询 UDDI 注册中心，为请求者寻找满足请求的服务；（发现）

3）Web 服务中介者向 Web 服务请求者返回满足条件的 Web 服务描述信息，该描述信息用 WSDL 写成，各种支持 Web 服务的机器都能阅读；（发现）

4）利用从 Web 服务中介者返回的描述信息生成相应的 SOAP 消息，发送给 Web 服务提供者，以实现 Web 服务的调用；（绑定）

5）Web 服务提供者按 SOAP 消息执行相应的 Web 服务，并将服务结果返回给 Web 服务请求者。（绑定）

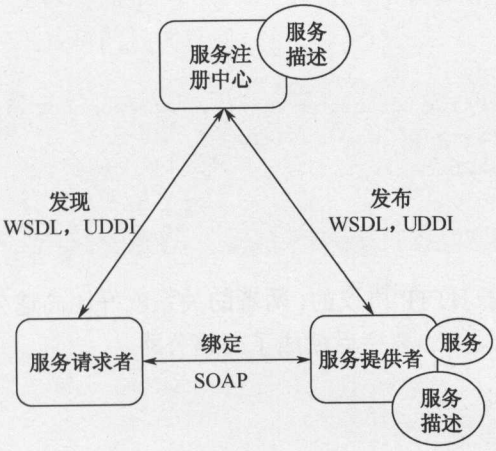


图 11-1　Web 服务

8. Web Service 的技术原理

（1）SOAP 协议

Web Service 采用 HTTP 协议传输数据，采用 XML 格式封装数据（即 XML 中说明调用远程服务对象的哪个方法，传递的参数是什么，以及服务对象的返回结果是什么）。Web Service 通过 HTTP 协议发送请求和接收结果时，发送的请求内容和结果都采用 XML 格式封装，并增加一些特定的 HTTP 消息头，以说明 HTTP 消息的内容格式，这些特定的 HTTP 消息头和 XML 内容格式就是 SOAP 协议（simple object access protocol，简单对象访问协议）。

SOAP 协议= HTTP 协议+ XML 数据格式。

SOAP 消息格式代码如下：

```
1       <?xml version="1.0"?>
2       <soap:Envelope
3       xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
```

```
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Header>
      <m:Trans xmlns:m="http://www.w3schools.com/transaction/"
      soap:mustUnderstand="1">234
      </m:Trans>
    </soap:Header>


    <soap:Body>
      <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
        <m:Item>Apples</m:Item>
      </m:GetPrice>
    </soap:Body>
    </soap:Envelope>
```

SOAP 协议是基于 HTTP 协议的，两者的关系就好比高速公路是基于普通公路改造的，在一条公路上加上隔离栏后就成了高速公路。

（2）WSDL 文件

Web Service 的服务器端要通过一个 WSDL 文件来说明自己家里有哪些服务可以对外调用，服务是什么（服务中有哪些方法，方法接受的参数是什么，返回值是什么），服务的网络地址用哪个 URL 地址表示，服务通过什么方式来调用。

WSDL 是基于 XML 格式的，它是 Web Service 客户端和服务器端都能理解的标准格式，其中描述的信息可以分为 what、where、how 等部分。

WSDL 文件保存在 Web 服务器上，通过一个 URL 地址就可以访问到它。客户端要调用一个 Web Service 服务之前，要知道该服务的 WSDL 文件的地址。

Web Service 服务提供商可以通过以下两种方式来暴露它的 WSDL 文件地址。

- 注册到 UDDI 服务器，以便被人查找；
- 直接告诉给客户端调用者，例如，在自己的网站给出信息或邮件告诉使用者。

9. Web Service 客户端的实现技术

调用 Web Service 服务的客户端的方法大致有三种，分别是：Axis、XFire 和 CXF。这三种技术的实现分别如下。

（1）使用 Axis 调用

如果提供的远程服务方法传入的参数都是简单类型，可以不用生成客户端代码，直接手动编写调用远程服务代码即可。

查看 WSDL 描述文件，wsdl：portType 暴露的是远程接口名称；wsdl：operation 对应的 name 为远程接口暴露的方法；一般情况下，wsdl：definitions 定义的 targetNamespace 对应的是接口的包的反向。

```
import java.net.URL;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

public class AxisClientSample {
 public static void main(String[] args) throws Exception {
  String[] recipients = new String[]{}; //收件人
  String strSubject = " ";                //主题
  String strContent = " ";       //内容
  String[] ccRecipients = new String[]{}; //抄送人
  String[] bccRecipients = null;     //密送人

  String endPoint = "http://service_host:port/appProject/services/
XXXXService";
  Service server = new Service();
  Call call = (Call) server.createCall();
  call.setTargetEndpointAddress(new URL(endPoint));
  //call.getMessageContext().setUsername("username");
  //如果远程方法需要用户名和密码验证时
  //call.getMessageContext().setPassword("password");
  String resp= (String) call.invoke("exposeMethod", new Object[]
{recipients, strSubject,strContent, ccRecipients,
 bccRecipients});
  System.out.println(resp);
 }
}
```

如果参数是复杂类型，使用 Axis 对 WSDL 文件生成客户端代码，使用 Axis 的 WSDL2JAVA 组件生成客户端代码，编写调用代码如下：

```
//获取远程服务
 XXXXServiceService service = new XXXXXLocator();
//XXXXService 是远程服务接口类
 XXXXService stub= service.getXXXXService();
```

```
    //构建参数 args，实际调用远程服务方法
    stub.exposeMethod(args);
```

（2）XFire 调用

方法与 Axis 类似，简单参数可以直接手写调用代码，复杂类型可以生成客户端代码或者手写。

可以根据 WSDL 描述得到对应的包名、接口名称和方法名，手动编写接口类，比如 XXXService，代码如下：

```
import java.net.MalformedURLException;
import org.codehaus.xfire.XFireFactory;
import org.codehaus.xfire.client.XFireProxyFactory;
import org.codehaus.xfire.service.Service;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;

public class XFireClientSample {
 public static void main(String[] args) {
  Service srModel = new ObjectServiceFactory().create(XXXService.class);
  XFireProxyFactory factory = new XfireProxyFactory
(XFireFactory.newInstance().getXFire());
  String endPoint = "http://service_host:port/app/services/XXXService";
  try {
   XXXService service = (XXXService) factory.create(srModel, endPoint);
   String[] recipients = new String[] {};
   String strSubject = "";
   String strContent = "";
   String[] ccRecipients = new String[] {}; // 抄送人
   String[] bccRecipients = null; // 密送人
   System.out.print("提交返回：" + service.getMethod(recipients,
strSubject, strContent, ccRecipients, bccRecipients));
  } catch (MalformedURLException e) {
   e.printStackTrace();
  }
 }
}
```

（3）使用 CXF 调用

代码如下：

```
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import test.demo.service.XXXService;
```

```
public class CXFClientSample {
 public static void main(String[] args) throws Exception{
  JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
  factory.setAddress("http://remote_host:port/service/XXXService");
  factory.setServiceClass(XXXService.class);
  XXXService service = (XXXService) factory.create();
  String username="sample";
  String response = service.executeMethod(username);
  System.out.println(response);
 }
}
```

10. Web Service 的技术实例

最初的 Web Service 通常是可以方便地并入应用程序的信息来源，如股票价格、天气预报、体育成绩等。

利用 Web Service 技术，可以构建更强大的应用程序。

例如，用户可以开发一个采购应用程序，以自动获取来自不同供应商的价格信息，从而使用户可以选择供应商，提交订单，然后跟踪货物的运输，直至收到货物。而供应商的应用程序除了在 Web 上提供服务外，还可以使用 Web Service 检查客户的信用、收取货款，并与货运公司办理货运手续。

以下将讲述两个技术案例，分别是：传输复杂对象的实现和跨多个 Web Service 进行管理 Session。

（1）Web Service 传输复杂对象类型的实现

复杂对象类型包含：字节数组、返回一维 int 数组、二维 String 数组及自定义 JavaBean 对象。传输复杂对象类型的 Web Service 的服务器端的代码如下：

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;
import data.User;

/**
 *传输复杂对象类型的 Web Service
 * @package
```

```
 */
public class ComplexTypeService {

    public String upload4Byte(byte[] b, int len) {
        String path = " ";
        FileOutputStream fos = null;
        try {
         String dir = System.getProperty("user.dir");
         File file = new File(dir + "/" + new Random().nextInt(100)
                              + ".jsp");
         fos = new FileOutputStream(file);
         fos.write(b, 0, len);
         path = file.getAbsolutePath();
         System.out.println("File path: " + file.getAbsolutePath());
        } catch (Exception e) {
         e.printStackTrace();
        } finally {
         try {
             fos.close();
         } catch (IOException e) {
             e.printStackTrace();
         }
        }
        return path;
    }

    public int[] getArray(int i) {
        int[] arr = new int[i];
        for (int j = 0; j < i; j++) {
         arr[j] = new Random().nextInt(1000);
        }
        return arr;
    }

    public String[][] getTwoArray() {return new String[][]
{ { "中国", "酒吧" }, { "日本", "北海道" }, { "中国", "影院", "南京" } };
    }

    public User getUser() {
        User user = new User();
        user.setAddress("china");
        user.setEmail("tom@XXX.com");
        user.setName("tom");
        user.setId(22);
```

```
        return user;
    }
}
```

上面的 Web Service 服务分别是传递字节完成数据上传、返回一维 int 数组、二维字符串数组，以及 User JavaBean 对象。

User Bean 代码如下：

```
package data;

import java.io.Serializable;

/**
 * User Entity
 */
public class User implements Serializable {
    private static final long serialVersionUID = 677484458789332877L;
    private int id;
    private String name;
    private String email;
    private String address;



    @Override
    public String toString() {
        return this.id + "#" + this.name + "#" + this.email + "#" +
this.address;
    }
}
```

值得注意的是，这个 User 对象的 package 是 data，如果是其他的 package，就需要在 tomcat→webapps→axis2→WEB-INF 目录下创建一个 data 目录，和 User 对象的目录保持一致，否则 Web Service 将会出现 ClassNotFontException 异常，Tomcat 会重启。

利用客户端的 Axis 开发技术，编写调用 Web Service 服务的客户端代码如下：

```
package com.tom.service;

import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.IOException;
import javax.xml.namespace.QName;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.rpc.client.RPCServiceClient;
import com.tom.entity.User;

/**
 *复杂类型数据 Web Service 客户端的代码
* @file ComplexTypeServiceClient.java
 */
public class ComplexTypeServiceClient {

    public static void main(String[] args) throws IOException {
        RPCServiceClient client = new RPCServiceClient();
        Options options = client.getOptions();
        String address = "http://localhost:8080/axis2/services/
ComplexTypeService";
        EndpointReference epr = new EndpointReference(address);
        options.setTo(epr);

        QName qname = new QName("http://ws.apache.org/axis2", "
upload4Byte");
        String path = System.getProperty("user.dir");
        File file = new File(path + "/WebRoot/index.jsp");
        FileInputStream fis = new FileInputStream(file);
        int len = (int) file.length();
        byte[] b = new byte[len];
        int read = fis.read(b);
        //System.out.println(read + "#" + len + "#" + new String(b));
        fis.close();
        Object[] result = client.invokeBlocking(qname, new Object[] { b,
len }, new Class[] { String.class });
        System.out.println("upload:" + result[0]);

        qname = new QName("http://ws.apache.org/axis2", "getArray");
        result = client.invokeBlocking(qname, new Object[] { 3 }, new Class[]
{ int[].class });
        int[] arr = (int[]) result[0];
        for (Integer i : arr) {
         System.out.println("int[] :" + i);
        }
```

```
            qname = new QName("http://ws.apache.org/axis2", "getTwoArray");
            result = client.invokeBlocking(qname, new Object[] {}, new Class[]
{ String[][].class });
            String[][] arrStr = (String[][]) result[0];
            for (String[] s : arrStr) {
             for (String str : s) {
                 System.out.println("String[][]: " + str);
             }
            }

            qname = new QName("http://ws.apache.org/axis2", "getUser");
            result = client.invokeBlocking(qname, new Object[] {}, new Class[]
{ User.class });
            User user = (User) result[0];
            System.out.println("User: " + user);
        }
    }
```

客户端代码运行后的结果是：

```
upload:D:\tomcat-6.0.28\bin\28.jsp
int[] :328
int[] :241
int[] :932
String[][]: 中国
String[][]: 酒吧
String[][]: 日本
String[][]: 北海道
String[][]: 中国
String[][]: 影院
String[][]: 南京
User: 22#tom#tom@XXX.com#china
```

（2）跨多个 Web Service 管理 Session

当有多个Web Service时，若要管理它的Session，此时要依靠ServiceGroupContext保存 Session 信息；并且，在客户端需要开启对 Session 的管理，即需要在客户端的程序中设置 options.setManageSession(true)。

1）多个 Web Service 的 Session 管理代码如下：

```
package com.tom.service;

import org.apache.axis2.context.MessageContext;
```

```
import org.apache.axis2.context.ServiceGroupContext;

/**
 *管理多个会话 Session 信息
 * @file LoginSessionService.java
 * @package com.tom.service
*/
public class LoginSessionService {
    public boolean login(String userName, String password) {
        MessageContext context =
MessageContext.getCurrentMessageContext();
        ServiceGroupContext ctx = context.getServiceGroupContext();
        if ("yangming".equals(userName) && "123456".equals(password)) {
         ctx.setProperty("userName", userName);
         ctx.setProperty("password", password);
         ctx.setProperty("msg", "登录成功");
         return true;
        }
        ctx.setProperty("msg", "登录失败");
        return false;
    }

    public String getLoginMessage() {
        MessageContext context =
MessageContext.getCurrentMessageContext();
        ServiceGroupContext ctx = context.getServiceGroupContext();
        return ctx.getProperty("userName") + "#" + ctx.getProperty("
msg");
    }
}
```

另外，还需要用一个 Service 来查询 session 的信息，SearchService 的代码如下：

```
package com.tom.service;

import org.apache.axis2.context.MessageContext;
import org.apache.axis2.context.ServiceGroupContext;

/**
 *查找多服务 Session 会话中的消息
 * @file SearchSessionServcie.java
 * @package com.tom.service
 */
public class SearchSessionServcie {
```

```
    public String findSessionMessage(String key) {
        MessageContext mc = MessageContext.getCurrentMessageContext();
        ServiceGroupContext ctx =  mc.getServiceGroupContext();
        if (ctx.getProperty(key) != null) {
            return "找到<" + key + ", " + ctx.getProperty(key) + ">";
        } else {
            return "没有找到<" + key + ">";
        }
    }
}
```

2）编写 services.xml 来发布这两个服务。这一次是用一个 services.xml 文件配置两个 service，同时发布两个服务。XML 代码如下：

```
<serviceGroup>
    <service name="LoginSessionService" scope="application">
        <description>
         Session Sample
        </description>
        <parameter name="ServiceClass">
         com.tom.service.LoginSessionService
        </parameter>
        <messageReceivers>
         <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
             class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
         <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
             class="
org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
        </messageReceivers>
    </service>

    <service name="SearchSessionService" scope="application">
        <description>
         Session Sample
        </description>
        <parameter name="ServiceClass">
         com.tom.service.SearchSessionServcie
        </parameter>
        <messageReceivers>
         <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
             class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
         <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
             class="
org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
```

```
            </messageReceivers>
        </service>
    </serviceGroup>
```

3）发布完成后，可以通过 http：//localhost:8080/axis2/services/listServices 查看发布的 Web Service 服务，编写客户端的测试代码如下：

```
    package com.tom.service;

    import javax.xml.namespace.QName;
    import org.apache.axis2.AxisFault;
    import org.apache.axis2.addressing.EndpointReference;
    import org.apache.axis2.client.Options;
    import org.apache.axis2.rpc.client.RPCServiceClient;

    /**
     *多会话 Session 管理，Web Service 客户端请求代码
     * @file LoginSessionServiceClient.java
     * @package com.tom.service
     */
    public class LoginSessionServiceClient {

        public static void main(String[] args) throws AxisFault {
            String target = "
http://localhost:8080/axis2/services/LoginSessionService";
            RPCServiceClient client = new RPCServiceClient();
            Options options = client.getOptions();
            options.setManageSession(true);

            EndpointReference epr = new EndpointReference(target);
            options.setTo(epr);

            QName qname = new QName("http://service.tom.com", "login");
            //指定调用的方法和传递参数数据，及设置返回值的类型
            Object[] result = client.invokeBlocking(qname, new Object[] {"
yangming", "123456" }, new Class[] { boolean.class });
            System.out.println(result[0]);

            qname = new QName("http://service.tom.com", "getLoginMessage");
            result = client.invokeBlocking(qname, new Object[] { null }, new
Class[] { String.class });
            System.out.println(result[0]);
```

```
            target = "
http://localhost:8080/axis2/services/SearchSessionService";
            epr = new EndpointReference(target);
            options.setTo(epr);

            qname = new QName("http://service.tom.com", "
findSessionMessage");
          result = client.invokeBlocking(qname, new Object[] { "userName" },
new Class[] { String.class });
            System.out.println(result[0]);
            qname = new QName("http://service.tom.com", "
findSessionMessage");
            result = client.invokeBlocking(qname, new Object[] { "msg" }, new
Class[] { String.class });
            System.out.println(result[0]);

            qname = new QName("http://service.tom.com", "
findSessionMessage");
          result = client.invokeBlocking(qname, new Object[] { "password" },
new Class[] { String.class });
            System.out.println(result[0]);
        }
    }
```

运行后的结果如下：

```
true
yangming#登录成功
找到<userName, yangming>
找到<msg, 登录成功>
找到<password, 123456>
```

11.2 高并发

1. 高并发架构设计考虑的因素

- 架构设计的目标：更充分地利用资源；提升性能/并发能力。
- 架构设计的方法：对系统关键结点及上下文充分了解；充分理解系统业务模型和特点。
- 架构设计的手段：系统资源间的巧妙协作；优化资源配比。

2. 高并发架构设计可采用的技术

- 减少请求处理：裁剪请求、请求合并、等待列表（waiting list）。
- 减少重复计算：结果缓存（cache）/共享内存/享元模式减少系统消耗；异步/同步，select/epoll；使用线程或者进程；内存分配优化；资源预取。
- 减少数据传输：协议优化 json/protobuf/…；数据压缩；共享词典。

3. 高并发架构设计的最终目的

极限负载下保证系统的稳定输出。

为了实现这个目的，不能忘记的技术有：主动过载保护；容灾机制，如容灾缓存（cache）；队列模型优化；等待列表（waiting list）。

典型的高并发服务器的服务架构如图 11-2 所示。

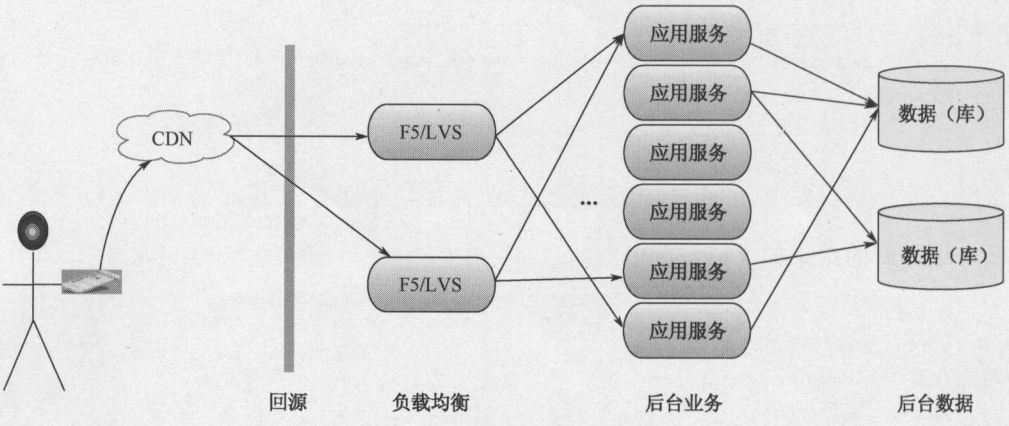


图 11-2　高并发的服务架构

简单地说，在图 11-2 中，CDN（内容分发网络）就是网络加速，分为静态加速和动态加速。静态加速就是结点缓存，用户实现就近访问最近的结点缓存。动态加速就是用户实现访问路径加速。可以这样理解：CDN = 镜像 + 智能解析。

镜像就是静态缓存/静态加速，把网站静态的东西镜像到各结点服务器上。

智能解析就是动态加速，是一种智能定向。比如：河北网通的用户访问网站，就解析到最近的网通结点上。广东的电信用户访问就解析到最近的电信结点上。

当用户在静态缓存中不能得到需要的数据时，就击穿 CDN，到服务器集群中寻找数据。

在每一个集群内，实现负载均衡管理，将用户的请求导向负载不太大的服务器。

应用服务通过用户的请求，得到用户所需要的数据。

4. 高并发服务在设计时需要注意的短桶原理

并发能力是所有服务对外的整体表现，所以是由短板结点的服务能力决定的短板结点可能出现在如下位置。

- 接入集群的出口/入口带宽；
- 核心业务模块的性能（即计算能力、内存模型）；
- 接入服务的并发连接数（即服务模型）。

（1）从计算能力角度考虑

从内部的请求来看，网络服务提供计算力的架构如图 11-3 所示。从图 11-3 可知，从计算能力上说，高并发的实现要考虑如下因素。

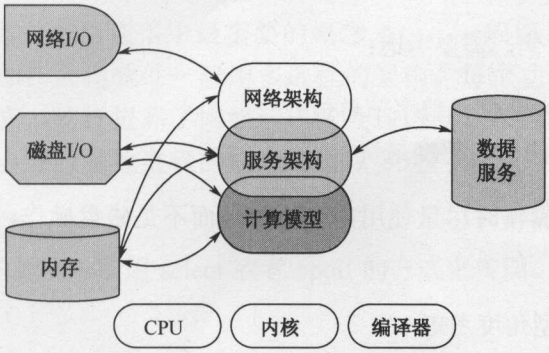


图 11-3　网络服务的计算力架构

1）对数据密集型（在线服务）来说，需要考虑：

- 访问模型：实时读写/实时读，批量写/流式读/批量读。
- 存储载体：远程数据服务/磁盘/SSD/内存。

2）对计算密集型来说，需要考虑：

- 串行依赖型；
- 交叉依赖性。

- 在复杂的数学计算中，除法的效率最低，应该尽量避免，尽量不要使用浮点数，比如，0.0001 这个浮点数可以乘以 10000 来得到一个整数，之后再进行计算。

（2）从内存模型角度考虑

从内存模型上说，高并发的实现要考虑如下因素。

1）对于内存池，需要考虑：

- 系统分配（伙伴系统）；
- 托管/预分配/allocator。

2）对于容器选型，需要考虑：

- 优先考虑：stl map/hash_map 等；
- stl 通用，但性能却不优。

3）对于访存优化，需要考虑：

- 以机器字为单位代替字节单位；
- 位图/数组的访问速度最快。

如果能在用户操作时尽量使用内存操作，而不是磁盘操作，则能增加网络请求的响应速度。

（3）从服务模型角度考虑

从整个用户服务的角度来看，制约服务能力的外部因素如图 11-4 所示。

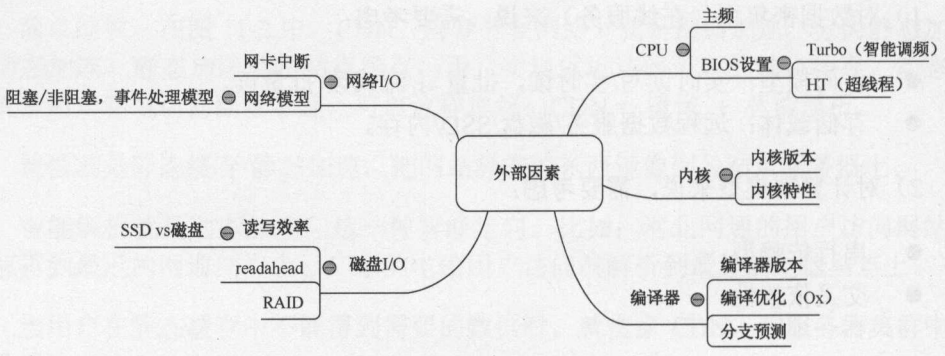


图 11-4　制约服务能力的外部因素

从图 11-4 可知，对于服务模型来说，高并发的实现要考虑如下因素。

1）对于线程模型，需要考虑：多线程、多进程、多线程与多进程混合。

2）对于网络架构，需要考虑：

- 异步事件、同步模型；
- 并发连接能力。

3）对于通信协议，需要考虑：

- 应用层通信协议；
- 单机进程间数据传递；
- 进程内数据传递/共享。

11.3 多线程与多进程

多线程和多进程是网络通信中最重要的概念之一，一般用于提高网络 I/O 请求的处理速度，比如：大型软件一般用多进程的架构，每个功能都有一个进程负责，进程间相对独立，这样提高了网络请求的处理速度、系统的运行速度。由于一个进程的崩溃不会导致其他进程的崩溃，所以多进程也变相地提高了系统的容灾性。

网络 I/O 请求的处理一般用 select 或者 epoll 的方式来实现。进程、线程与网络 I/O 请求的关系如下：

进程→线程→线程间用 select 或 epoll 进行 I/O 请求。

当然，一个进程里可以没有多线程，这时多进程与网络 I/O 请求的关系如下：

进程 → 进程间用 select 或 epoll 进行 I/O 请求。

线程和进程的区别有以下方面。

- 从创建过程来说，线程是轻量级进程，共享页表；进程是复制数据（文件描述符、页面项等）；

- 多线程的伸缩性强、稳定性差；多进程的伸缩性差、稳定性强（比如：提高了容灾性），更容易获得高性能。

11.3.1 多线程

多线程的架构如图 11-5 所示。

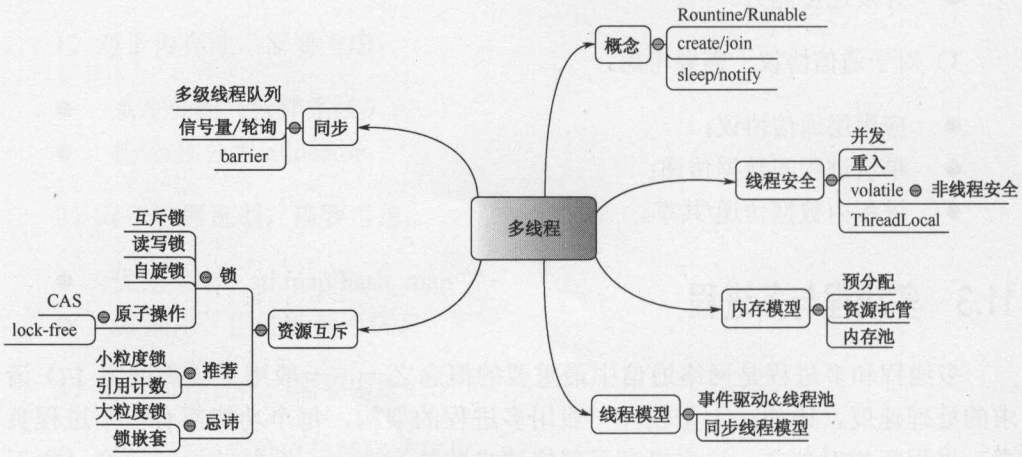


图 11-5　多线程的架构

多线程的一般模型如图 11-6 所示。

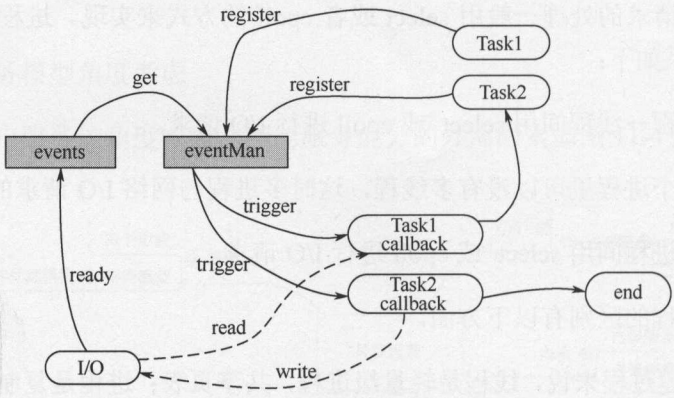


图 11-6　多线程的模型

从图 11-6 可知，在多线程的一般模型中，多线程任务的管理是通过对事件进行注册来实现的。

我们已经知道，高并发下可以采用的 I/O 通信机制有两种：select 与 epoll。这两种 I/O 模型因为都属于对请求的处理策略，所以都能适用于线程和进程的 I/O 请求处理。select 与 epoll 都属于多路 I/O 就绪通知，提供了对大量的文件描述符就绪检查的高性能方案，只是实现方式有所不同。

1. select

一个 select 系统调用来监视包含多个文件描述符的数组，当 select 函数运行结束时，该数组中就绪的文件描述符便会被内核修改标志位。

select 的跨平台做得很好，几乎每个平台都支持。但 select 也存在以下缺点。

1）单个进程能够监视的文件描述符的数量存在最大限制。

2）select 所维护的存储大量文件描述符的数据结构随着文件描述符数量的增长，其在用户态和内核的地址空间的复制所引发的开销也会呈线性增长。

3）由于网络响应时间的延迟，使得大量 TCP 连接处于非活跃状态，但调用 select() 还是会对所有的 Socket 进行一次线性扫描，从而就会造成一定的开销。

select 的优化形式：poll 本质上仍是 select，是对 select 的一种优化。poll 是 UNIX 沿用 select 自己重新实现了一遍，唯一解决的问题是 poll 没有最大文件描述符数量的限制。

2. epoll

epoll 具有以下两个优势来大幅度提升性能。

- 基于事件的就绪通知方式 select/poll 方式，进程只有在调用一定的方法后，内核才会对所有监视的文件描述符（事件）进行扫描，而 epoll 事件通过 epoll_ctl()注册一个文件描述符，一旦某个文件描述符就绪，内核就会采用类似 call back 的回调机制，迅速激活这个文件描述符，epoll_wait()便会得到通知；
- int epoll_wait（int epfd, struct epoll_event * events, int maxevents, int timeout）函数返回需要处理的事件（即：epoll 专用的文件描述符）数目，如返回 0 表示已超时。epoll_wait 函数返回的事件集合在 events 数组中，events 数组中实际存放的成员个数是函数的返回值。这里使用内存映射（mmap）技术，避免了复制大量的文件描述符带来的开销。

当然，epoll 也有一定的局限性，epoll 只有在 Linux 2.6 及以后的版本中才有其实现，而其他平台都没有，这和 apache（采用 select 方式处理网络 I/O 请求）这种优秀的跨平台服务器显然有些背道而驰。

11.3.2 多进程

多进程的典型模型如图 11-7 所示。

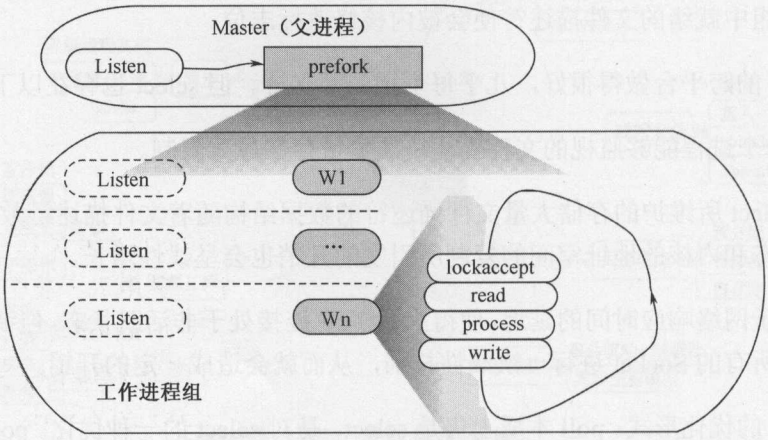


图 11-7 多进程的模型

多进程间共享可以采用很多技术，比如：IPC，即管道、FIFO 或消息队列。但由于两个进程交换信息时，这些技术都需要先经过内核来传递，所以都不是最快的技术。

多个进程间通信一般采用的是 sharedmemory（共享内存）。一旦这样的内存区映射到共享它的进程的地址空间，这些进程间的数据传递就不再涉及内核。然而，往该共享内存区存放信息或从中取走信息的进程间需要某种形式的同步，即如下技术。

- 互斥锁；
- 条件变量；
- 读写锁；
- 记录锁；
- 信号量。

1. 两个典型的多进程与多线程的网络服务架构（Apache 与 Nginx）

尽管多线程和多进程都是在一台计算机内通信，但是多线程与多进程都可以与I/O 技术结合在一起用来处理网络请求。Apache 就是采用一个进程、多个线程（每个线程处理一个请求，线程间请求的轮询用 select 实现）来处理网络请求的。而 Nginx 是采用多个进程（一个父进程 listener，多个 worker 进程，每个进程对应多个线程）处理多个网络请求（请求间的轮询用 epoll 实现）的。

2. Apache 与 Nginx 的比较

Apache 与 Nginx 都是非常棒的网络服务架构，两者谁的性能更高效取决于其服务器的并发策略和其面对的场景。

（1）并发策略

我们目前使用的 Apache 是基于一个线程处理一个请求的非阻塞 I/O 并发策略。这种方式允许一个进程中通过多个线程来处理多个连接，其中每个线程处理一个连接。Apache 使用其 worker 模块实现这种方式，目的是减少 perfork 模式中太多进程的开销，使得 Apache 可以支持更多的并发连接。至于非阻塞 I/O 的实现，是通过一个子进程负责 accept()，一旦接收到连接，便将任务分配给合适的 worker 线程。由于 Apache 的线程使用的是内核进程调度器管理的轻量级进程，因此，与 perfork 模式比较，进程上下文切换的开销依然存在，性能提升不是很明显。

从 I/O 的层面来说，这种方式仍然类似 poll 的实现方式，本质上是 select 的请求轮询方式。

而 Nginx 使用的是一个 worker 进程处理多个线程连接、非阻塞 I/O 模式，这种模式最特别的是设计了独立的 listener 进程，专门负责接收新的连接，再分配给各个 worker 进程。

而 I/O 模型层面，Nginx 选择 epoll，该方式高效主要在于其基于事件的就绪通知机制，在高连接数的场景下，epoll 通知方式更具优势。另外，epoll 方式只关注活跃连接，而不像 select 方式需要扫描所有的文件描述符，这样在大量连接的场景下，epoll 方式的优势会更加明显。

（2）面对的场景

如果是动态网站，网站的大部分内容都需要动态获取、计算和输出，因此，响

应时间普遍在 100ms 以上，响应时间较长，服务器必将创建更多的连接处理这些请求。在 TCP 监控当中，可以看到由于 TCP 协议特有的特性，服务器端主动关闭一个连接后，连接会进入等待超时的状态，且此状态会持续 2MSL（即两倍的数据包最大生存时间，这个时间长短与操作系统有关，一般会在 1 到 4 分钟）。因此，服务器端会保留一定量超时的 time_wait 连接。管理大量的连接也会对服务器造成一定的成本，而 epoll 在多连接并发处理和管理方面都比 select 具有很大的优势。这也正是高并发、高连接的互联网网站（如淘宝）大量使用 Nginx 服务器的原因所在。

LBS 各领域常用的开发资源（常用库及 API）

搜索引擎（百度、谷歌）：若有问题，多上搜索引擎。搜索引擎是程序员最重要的工具，它能让程序员站在巨人的肩膀上看“世界”。

几何处理：GEOS、Spatial

图像处理：OPENCV、GDAL、Magick

三维渲染：OPENGL

点云处理：PCL

地图 API：高德地图 API、百度地图 API

多进程：nginx

多线程：IOCP（Windows）、epoll（Linux）

高速缓存：redis、memcached

内存池：tcmalloc

RPC 协议：protobuf、thrift

本书主要术语的定义或说明

本书涉及的一些基本定义如下。

（1）数据

指对外界事物的一种抽象模拟。

（2）数据格式

指解决应用的某种需求的一种特定解决方案，比如：为了更快速地进行查找，人们发明了二叉树、红黑树。为了能快速查找大规模的数据，人们发明了大型数据库（B 树）。

（3）应用软件

应用软件是人们为了实现某种目的，而对数据进行操作、交互的界面化工具。

（4）UI

UI 是应用软件的外部表现形式，往往体现为使用者可以直接感受到某种界面。

（5）用户输入

用户输入是应用软件的输入数据，以鼠标点击、语音输入等形式来体现。

（6）引擎

引擎是应用软件中利用一定的用户输入得到特定输出的程序实现。

（7）关于数据结构方面的定义

- B-树：平衡树；
- B 树：Binary 树，即二分树；
- Hash 表：散列表。

（8）与位置应用相关的定义

- App：应用程序；
- LBS：基于位置的服务；
- O2O（online 2 offline）：即在线/离线、线上到线下；
- GIS：地理信息系统。

（9）关于地图模块的定义

- BMD：基本显示模块；
- 3D：三维物体，往往指三维建筑物；
- POI：兴趣点；
- ROUTE：道路，往往指与道路数据有关的操作；
- JV：路口指示图；
- ADMIN：行政区划；
- VOICE：导航语音；
- ORTHO：卫星正交影像；
- DTM：数字高程图；
- TMC：实时交通信息。

（10）关于地图数据格式的定义

- NDS：欧洲高端车系的物理格式导航数据标准；
- KIWI：日系车的物理格式导航数据标准；
- GDF：一种交换格式的数据厂商的母库格式标准（欧洲和亚洲中国的数据制作厂商应用普遍）；
- RDF：日本三菱的一种交换格式的数据标准（中国、日本的导航数据制作厂商偶尔使用）；
- MIF：中国主流的交换格式数据（高德、四维图新），广泛用于手机导航。

（11）关于地图空间索引方面的定义

- K-d 树索引（K dimension tree）：是一种二叉树的方法，先对 x/y 进行二分，之后对 y/x 进行二分，再对 x/y 进行二分……
- 四叉树索引/网格空间索引：本质上属于 K-d 树索引体现在空间上的一种形式；
- R 树空间索引：以最小边界矩形（简称 MBR）递归的对数据集空间按照“面积”规则进行划分。

（12）关于图像处理方面的定义

- RANSAC：利用随机采样得到边线检测的方法；
- HOUGH 变换：将像素空间转为参数空间，从而对图像进行直线或圆等形状的检测；
- CRC（Cyclic Redundancy Check）：循环冗余校验码，一种对原始数据的抽取特征（一般小于 32 位）的方法；
- MD5：一种对原始数据的抽取 128 位特征的方法。

（13）关于搜索方面的定义

- FTS：full content search（全文搜索）；
- NVC：next valid character（下一个有效字符）。

（14）关于显示方面的定义

LOD：level of display（分层细节显示）。

（15）关于网络传输方面的定义

- TCP/IP 协议：目前主流的网络通信技术；
- UDP 协议：一种具有多播功能的通信技术，广泛用于网络流媒体技术。

（16）关于 Web Service 方面的定义

- XML（Extensible Markup Language）：扩展型可标记语言。面向短期的临时数据处理，面向万维网络，是 SOAP 的基础；
- SOAP（Simple Object Access Protocol）：简单对象存取协议，是 XML Web Service 的通信协议。当用户通过 UDDI 找到 WSDL 描述文档后，可以通过 SOAP 调用建立的 Web 服务中的一个或多个操作。SOAP 是 XML 文档形式的调用方法的规范，它可以支持不同的底层接口，像 HTTP(S)或者 SMTP；
- WSDL（Web Services Description Language）：是一个 XML 文档，用于说明一组 SOAP 消息以及如何交换这些消息。大多数情况下，由软件自动生成和使用；
- UDDI（Universal Description, Discovery, and Integration）：是主要针对 Web 服务供应商和使用者的一个新项目。在用户能够调用 Web 服务之前，必须确定这个服务内包含哪些商务方法，找到被调用的接口定义，还要在服务

器端来编制软件，UDDI 是一种根据描述文档来引导系统查找相应服务的机制。UDDI 利用 SOAP 消息机制（标准的 XML/HTTP）来发布、编辑、浏览和查找注册信息。它采用 XML 格式来封装各种不同类型的数据，并且发送到注册中心或者由注册中心返回需要的数据。

（17）关于高并发方面的定义

- IOCP：Windows 下的高并发技术；
- epoll：Linux 下的高并发技术；
- CDN：内容分发网络，是一种网络加速的技术。

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

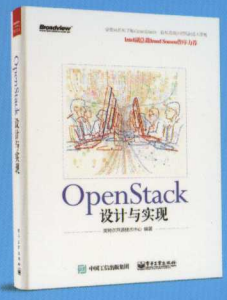
提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。

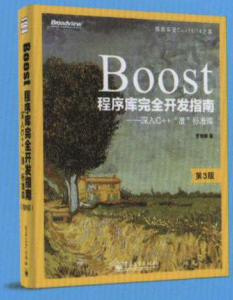
联系我们

博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/
投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com　@博文视点Broadview　　微信公众账号　博文视点Broadview

好书力荐





拒绝堆砌臃肿，支持纯正原创

出版事宜请关注 新浪微博 weibo.com @ 半亩方塘 _

投稿邮箱：sxy@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

LBS核心技术揭秘

业内专家力荐

本书如同庖丁解牛一般地阐述LBS核心技术，让你分分钟成为LBS专家。

阿里资深工程师 陈岳

谁看谁知道！本书是移动互联网时代核心技术解密！

阿里资深工程师 邹剑章

LBS是互联网行业的兵家必争之地，本书更是LBS大数据产业里最璀璨的宝藏。

阿里资深工程师 王奇

本书把LBS开发所需的知识串联起来，既实用，又深刻，并补充了市场上LBS技术开发教材的短缺。

阿里资深工程师 逯志欣

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：孙学瑛
责任编辑：李利健
封面设计：李　玲

上架建议：位置定位服务

ISBN 978-7-121-26214-2

9 787121 262142 >

定价：69.00元